scTDC Python SDK

Release 1.4.0

Surface Concept GmbH

Nov 20, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This Python SDK provides a Python 3 interface to the scTDC library. It has been tested with Python version 3.8.10. The products supported by this SDK are delay-line detectors (DLDs), stand-alone time-digital converters (TDCs), and Recon*Flex*™ cameras. In the current version, Recon*Flex*™ camera support does not cover reading of blob data.

1.1 SDK contents

This SDK comprises

1. the Python module *scTDC* contained in the file `scTDC.py`
2. code examples (files *example_*.py*)
3. this document

For running applications based on this SDK, you will also need files from the regular (C/C++) scTDC SDK and from your demo software folder.

1.2 Prerequisites

The scTDC.py needs additional files, not included here, to work. At the point, where your Python code does `import scTDC`, none of these files are checked at first, so this step should always succeed. There are two further stages where your code may run into errors due to missing files:

- At the point where your code creates an *scTDClib* object, either directly or by creating a *Device* object. During this stage, the following libraries are required:
  - `scTDC1.dll`
  - `pthreadVC2.dll`

  These libraries require that the Microsoft Visual C++ redistributable version 2015-2022 is installed, see
  - https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist

  or these direct permalinks:
  - https://aka.ms/vs/17/release/vc_redist.x86.exe (32 bit version)
  - https://aka.ms/vs/17/release/vc_redist.x64.exe (64 bit version)

  If these libraries cannot be loaded, an *OSError* exception is thrown.
- At the point where your code initializes the device via one of the respective functions (such as *Device.initialize*):

  - libraries related to the hardware interface (one of the following combinations)
    * `scDeviceClass60.dll`, `okFrontPanel.dll` (for USB TDCs)
    * `scDeviceClass450.dll` (for Ethernet interface)
    * `scDeviceClass6010.dll`, `FTD3XX.dll` (for ReconFlex™ cameras)
  - if you have a ReconFlex™ camera product, the library `para3.dll`, (sometimes it must be renamed to `para30.dll`)
  - the configuration file `tdc_gpx3.ini`
  - USB TDCs that came with a `scDeviceClass60.dll` require a firmware file with the `*.bit` file name extension.
  - `cal_i.tif`, `cal_xyt.txt`, if any of these were present in the demo software folder shipped on USB pendrive together with your product

  Missing libraries from the above list result in an error code `-12` (`SC_TDC_ERR_DEVCLS_LD`) returned by the initialization function.

- We recommend to place all files mentioned in this section into one folder. The Python application can then use `os.chdir(...)` to change the current working directory to this folder, then create the object to the interface class of your choice (*Device*, *Camera*, or others), then initialize the device. Afterwards, the working directory can be changed back to the original location.

1.2.1 Linux notes

If you have placed dependent libraries into a folder somewhere under your home directory, you need to point the system linker to this folder using the `LD_LIBRARY_PATH` variable.

For example, if you have put the libraries into `/home/user/abc`, you can start your python application via:

```
LD_LIBRARY_PATH="/home/user/abc" python3 my_application.py
```

1.3 Python module dependencies

The `scTDC` Python module makes use of the following modules.

- `numpy` (tested with version `1.21.4`)

The examples additionally require

- `matplotlib` (tested with version `3.1.2`)

These dependencies can usually be installed as follows:

```
python -m pip install --user numpy
python -m pip install --user matplotlib
```

You may have to replace `python` by the full path to your Python interpreter executable on Windows, or by `python3` on Linux. If you are using an Anaconda Python distribution, refer to the Anaconda documentation for installation of packages, instead.

CHAPTER

TWO

EXAMPLES OVERVIEW

2.1 TDC and DLD applications

The following examples are related to processing of event data (in a list-of-events form):

- **example_buffered_data_callbacks.py** a very basic example, using the BUFFERED_DATA_CALLBACKS pipe, which has been introduced in scTDC library version 1.3010.0 to enable better performance in python than the USER_CALLBACKS pipe.
- **example_buffered_data_callbacks2.py** computation of a histogram from the event data in python code using numpy and on-screen visualization using matplotlib
- **example_buffered_data_callbacks3.py** much improved performance of the previous example
- **example_buffered_data_callbacks4.py** data export to a text file for DLD events
- **example_buffered_data_callbacks5.py** data export to a text file for TDC events
- **example_user_callbacks_interface.py** Not recommended anymore.

The following examples demonstrate reading of histograms that are provided by the scTDC library

- **example_3d_pipe.py** A 3D histogram of DLD event count versus detector position x, y and time of DLD event
- **example_xy_xt_yt_pipes.py** 2D histograms for DLD events
- **example_tdc_histo_pipe.py** 1D histograms for stand-alone TDCs

2.2 Camera applications

- **example_camera.py**
  - setting of exposure parameters
  - starting a multiple-frame measurement
  - reading of image data and meta data
- **example_camera_blobs.py**
  - setting parameters related to blob-mode
  - reading of meta data and blob data

2.3 Integration with Qt5 (PyQt5)

- `example_pyqt5_tdc.py`

2.4 Miscellaneous

- **`example_eom_callback.py`** using the (general) end-of-measurement notification
- **`example_statistics_pipe.py`** read the "statistics" to obtain the number of pulses registered in the individual TDC channels and number of constructed DLD events (if applicable) during the last measurement. In case of delay-line detector applications, this is useful for diagnosis and to monitor the load on the MCP.

CHAPTER

THREE

API REFERENCE

The *scTDClib* class provides a low-level wrapper of functions from the underlying scTDC library. A few of the functions from this class may be interesting to the application, such as *sc_tdc_config_get_library_version* for querying the library version, or querying certain device properties. Other than that, this class is merely used to implement the higher-level interfaces which are recommended to be used primarily by the application developer:

- the *Device* class for access to pre-computed histograms from DLD and TDC data and statistics data. The *Device* class can create *Pipe* objects and return them to the application. For access to the data, methods from these *Pipe* objects are used.
- the *Camera* class in place of the *Device* class for camera applications.
- the *buffered_data_callbacks_pipe* class for access to TDC or DLD events in a list-of-events form.
- the *usercallbacks_pipe* class has been available for a longer time than the *buffered_data_callbacks_pipe* class and it maps the default list-of-events interface of the C/C++ scTDC SDK. However, the application may run into performance problems with the *usercallbacks_pipe* interface if TDC or DLD events are expected to be in the order of millions per second. The *buffered_data_callbacks_pipe* can be tuned to require less Python lines of code to be executed per second and therefore is able to handle higher event rates.

3.1 class scTDClib

class scTDC.**scTDClib**(*libfilepath=None*)

low-level wrapper of the C interface in the dynamically loaded library scTDC

__init__(*libfilepath=None*)

loads the library scTDC (scTDC1.dll or libscTDC.so) from hard disk and adds the correct signatures to the library functions so they can be used from Python.

Parameters **libfilepath** (*str*) – optionally specify the full path including file name to the shared library file, defaults to None

sc_tdc_init_inifile(*inifile_path='tdc_gpx3.ini'*)

Initializes the hardware and loads the initial settings from the specified ini file.

Parameters **inifile_path** (*str*) – the name of or full path to the configuration file, defaults to "tdc_gpx3.ini"

Returns Returns a non-negative device descriptor on success or a negative error code in case of failure. The device descriptor is needed for all functions that involve the initialized device

Return type int

sc_tdc_init_inifile_overrides(*inifile_path='tdc_gpx3.ini'*, *overrides=None*)

Initializes the hardware and loads the initial settings from the specified ini file. Enables overriding of parameters from the ini file without modification of the ini file on hard disk (the override entries reside in memory and are evaluated by the scTDC library).

Parameters

- **inifile_path** (*str*) – the name of or full path to the configuration file, defaults to "tdc_gpx3.ini"
- **overrides** (*list*) – a list of 3-tuples (section_name, parameter_name, parameter_value), where the section_name is specified without square brackets ([]). Spelling of names is case sensitive. defaults to [] (empty list)

Returns a non-negative device descriptor on success, or, a negative error code in case of failure. The device descriptor is needed for all functions that involve the initialized device

Return type int

sc_get_err_msg(*errcode*)

Returns an error message to the given error code.

Parameters **errcode** (*int*) – a negative error code returned by one of the library functions

Returns the error message describing the reason of the error code

Return type str

sc_tdc_config_get_library_version()

Query the version of the scTDC library.

Returns a 3-tuple containing the version separated into major, minor and patch parts, e.g. version 1.3017.5 becomes (1, 3017, 5)

Return type tuple

sc_tdc_deinit2(*dev_desc*)

Deinitialize the hardware.

Parameters **dev_desc** (*int*) – device descriptor as retrieved from *sc_tdc_init_inifile* or *sc_tdc_init_inifile_overrides*

Returns 0 on success or negative error code

Return type int

sc_tdc_start_measure2(*dev_desc*, *exposure_ms*)

Start a measurement (asynchronously/non-blocking)

Parameters

- **dev_desc** (*int*) – device descriptor as returned by one of the initialization functions
- **exposure_ms** (*int*) – The exposure time in milliseconds

Returns 0 on success or negative error code

Return type int

sc_tdc_interrupt2(*dev_desc*)

Interrupts a measurement asynchronously (non-blocking). Asynchronously means, the function may return before the device actually reaches idle state. *sc_tdc_set_complete_callback2* may be used to be notified when the device has stopped the measurement.

Parameters **dev_desc** (*int*) – device descriptor

Returns 0 on success or negative error code

Return type int

sc_pipe_open2(*dev_desc*, *pipe_type*, *pipe_params*)

Open a pipe for reading data from the device. The available pipe types with their corresponding pipe_params types are

- *TDC_HISTO* : *sc_pipe_tdc_histo_params_t*
- *DLD_IMAGE_XY* : *sc_pipe_dld_image_xyt_params_t*
- *DLD_IMAGE_XT* : *sc_pipe_dld_image_xyt_params_t*
- *DLD_IMAGE_YT* : *sc_pipe_dld_image_xyt_params_t*
- *DLD_IMAGE_3D* : *sc_pipe_dld_image_xyt_params_t*
- *DLD_SUM_HISTO* : *sc_pipe_dld_image_xyt_params_t*
- *STATISTICS* : *sc_pipe_statistics_params_t*
- *USER_CALLBACKS* : *sc_pipe_callback_params_t*
- *BUFFERED_DATA_CALLBACKS* : *sc_pipe_buf_callbacks_params_t*
- *PIPE_CAM_FRAMES* : None (no configuration parameters)
- *PIPE_CAM_BLOBS* : None (no configuration parameters)

Parameters

- **dev_desc** (*int*) – device descriptor
- **pipe_type** (*int*) – one of the pipe type constants
- **pipe_params** (*Any*) – various types of structures depending on pipe_type. If a structure is needed, it should be passed by value, not by pointer.

Returns a non-negative pipe handle on success or a negative error code

Return type int

sc_pipe_close2(*dev_desc*, *pipe_handle*)

Close a pipe.

Parameters

- **dev_desc** (*int*) – device descriptor
- **pipe_handle** (*int*) – the pipe handle as returned by *sc_pipe_open2*

Returns 0 on success or negative error code

Return type int

sc_pipe_read2(*dev_desc*, *pipe_handle*, *timeout*)

Read from a pipe. The functions waits until either data is available or the timeout is reached.

Parameters

- **dev_desc** (*int*) – device descriptor
- **pipe_handle** (*int*) – pipe handle as returned by *sc_pipe_open2*
- **timeout** (*int*) – the timeout in milliseconds

Returns a tuple containing the return code and a ctypes.POINTER to the data buffer

Return type tuple

sc_tdc_get_status2(*dev_desc*)

Query whether the device is idle or in measurement.

Parameters **dev_desc** (*int*) – device descriptor

Returns 0 (idle) or 1 (exposure) or negative error code

Return type int

sc_tdc_get_statistics2(*dev_desc*)

This function is deprecated. Use the statistics pipe, instead. This function is kept for older scTDC library versions.

sc_tdc_set_complete_callback2(*dev_desc*, *privptr*, *callback*)

Sets a callback to be notified about completed measurements or other events regarding the transition from measurement state to idle state.

Parameters

- **dev_desc** (*int*) – device descriptor
- **privptr** (*ctypes.POINTER(void)*) – a private pointer that is passed back into the callback
- **callback** (*function*) – the function to be called for notifications

Returns 0 on success or negative error code

Return type int

3.1.1 Pipe type constants

scTDC.**TDC_HISTO = 0**

pipe type, TDC time histogram for one TDC channel

scTDC.**DLD_IMAGE_XY = 1**

pipe type, image mapping the detector area

scTDC.**DLD_IMAGE_XT = 2**

pipe type, image mapping the detector x axis and the TDC time axis

scTDC.**DLD_IMAGE_YT = 3**

pipe type, image mapping the detector y axis and the TDC time axis

scTDC.**DLD_IMAGE_3D = 4**

pipe type, 3D matrix mapping the detector area the the TDC time axis

scTDC.**DLD_SUM_HISTO = 5**

pipe type, 1D histogram for DLDs, counts vs time axis

scTDC.**STATISTICS = 6**

pipe type, statistics data delivered at the end of measurements

scTDC.**USER_CALLBACKS = 10**

pipe type, TDC and DLD event data, slow in python

scTDC.**BUFFERED_DATA_CALLBACKS = 12**

pipe type, TDC and DLD event data, more efficient variant of USER_CALLBACKS

scTDC.**PIPE_CAM_FRAMES = 13**

pipe type, provides camera frame raw image data and frame meta data

scTDC.**PIPE_CAM_BLOBS = 14**

pipe type, provides camera blob coordinates

3.1.2 Data types for type parameters:

class scTDC.**sc3du_t**

```
class sc3du_t(ctypes.Structure):
 _fields_ = [("x",ctypes.c_uint),
             ("y",ctypes.c_uint),
             ("time", ctypes.c_uint64)]
```

class scTDC.**sc3d_t**

```
class sc3d_t(ctypes.Structure):
 _fields_ = [("x",ctypes.c_int),
             ("y",ctypes.c_int),
             ("time", ctypes.c_int64)]
```

class scTDC.**roi_t**

```
class roi_t(ctypes.Structure):
 _fields_ = [("offset", sc3d_t),
             ("size", sc3du_t)]
```

class scTDC.**sc_pipe_dld_image_xyt_params_t**

```
class sc_pipe_dld_image_xyt_params_t(ctypes.Structure):
 _fields_ = [("depth",    ctypes.c_int),
             ("channel",  ctypes.c_int),
             ("modulo",   ctypes.c_uint64),
             ("binning",  sc3du_t),
             ("roi",      roi_t),
             ("accumulation_ms", ctypes.c_uint),
             ("allocator_owner", ctypes.c_char_p),
             ("allocator_cb",    ALLOCATORFUNC)]
```

class scTDC.**sc_pipe_tdc_histo_params_t**

```
class sc_pipe_tdc_histo_params_t(ctypes.Structure):
 _fields_ = [("depth",     ctypes.c_int),
             ("channel",   ctypes.c_uint),
             ("modulo",    ctypes.c_uint64),
             ("binning",   ctypes.c_uint),
             ("offset",    ctypes.c_uint64),
             ("size",      ctypes.c_uint),
             ("accumulation_ms", ctypes.c_uint),
             ("allocator_owner", ctypes.c_char_p),
             ("allocator_cb", ALLOCATORFUNC)]
```

class scTDC.**sc_pipe_statistics_params_t**

```
class sc_pipe_statistics_params_t(ctypes.Structure):
 _fields_ = [("allocator_owner", ctypes.c_char_p),
             ("allocator_cb", ALLOCATORFUNC)]
```

class scTDC.**sc_pipe_callbacks**

```
class sc_pipe_callbacks(ctypes.Structure):
 _fields_ = [("priv",                ctypes.POINTER(None)),
             ("start_of_measure",    CB_STARTMEAS),
             ("end_of_measure",      CB_ENDMEAS),
             ("millisecond_countup", CB_MILLISEC),
             ("statistics",          CB_STATISTICS),
             ("tdc_event",           CB_TDCEVENT),
             ("dld_event",           CB_DLDEVENT)]

if os.name == 'nt':
    _FUNCTYPE = ctypes.WINFUNCTYPE
else:
    _FUNCTYPE = ctypes.CFUNCTYPE
CB_STARTMEAS  = _FUNCTYPE(None, ctypes.POINTER(None))
CB_ENDMEAS    = CB_STARTMEAS
CB_MILLISEC   = CB_STARTMEAS
CB_STATISTICS = _FUNCTYPE(None, ctypes.POINTER(None),
                          ctypes.POINTER(statistics_t))
CB_TDCEVENT   = _FUNCTYPE(None, ctypes.POINTER(None),
                          ctypes.POINTER(tdc_event_t), ctypes.c_size_t)
CB_DLDEVENT   = _FUNCTYPE(None, ctypes.POINTER(None),
                          ctypes.POINTER(dld_event_t), ctypes.c_size_t)
```

class scTDC.**sc_pipe_callback_params_t**

```
class sc_pipe_callback_params_t(ctypes.Structure):
 _fields_ = [("callbacks", ctypes.POINTER(sc_pipe_callbacks))]
```

class scTDC.**sc_pipe_buf_callbacks_params_t**

```
 class sc_pipe_buf_callbacks_params_t(ctypes.Structure):
_fields_ = [("priv",                  ctypes.POINTER(None)),
            ("data",                  CB_BUFDATA_DATA),
            ("end_of_measurement",    CB_BUFDATA_END_OF_MEAS),
            ("data_field_selection",  ctypes.c_uint),
            ("max_buffered_data_len", ctypes.c_uint),
            ("dld_events",            ctypes.c_int),
            ("version",               ctypes.c_int),
            ("reserved",              ctypes.c_ubyte * 24)]
```

3.1.3 Types returned when reading pipes (or getting callbacks from pipes):

class scTDC.**statistics_t**

```
class statistics_t(ctypes.Structure):
 _fields_ = [("counts_read", ctypes.c_uint * 64),
             ("counts_received", ctypes.c_uint * 64),
             ("events_found", ctypes.c_uint * 4),
             ("events_in_roi", ctypes.c_uint * 4),
             ("events_received", ctypes.c_uint * 4),
             ("counters", ctypes.c_uint * 64),
             ("reserved", ctypes.c_uint * 52)]
```

class scTDC.**tdc_event_t**

```
class tdc_event_t(ctypes.Structure):
 _fields_ = [("subdevice",     ctypes.c_uint),
             ("channel",       ctypes.c_uint),
             ("start_counter", ctypes.c_ulonglong),
             ("time_tag",      ctypes.c_ulonglong),
             ("time_data",     ctypes.c_ulonglong),
             ("sign_counter",  ctypes.c_ulonglong)]
```

class scTDC.**dld_event_t**

```
class dld_event_t(ctypes.Structure):
 _fields_ = [("start_counter",      ctypes.c_ulonglong),
             ("time_tag",           ctypes.c_ulonglong),
             ("subdevice",          ctypes.c_uint),
             ("channel",            ctypes.c_uint),
             ("sum",                ctypes.c_ulonglong),
             ("dif1",               ctypes.c_ushort),
             ("dif2",               ctypes.c_ushort),
             ("master_rst_counter", ctypes.c_uint),
             ("adc",                ctypes.c_ushort),
             ("signal1bit",         ctypes.c_ushort)]
```

class scTDC.**sc_pipe_buf_callback_args**

```
class sc_pipe_buf_callback_args(ctypes.Structure):
 _fields_ = [("event_index",        ctypes.c_ulonglong),
             ("som_indices",        ctypes.POINTER(ctypes.c_ulonglong)),
             ("ms_indices",         ctypes.POINTER(ctypes.c_ulonglong)),
             ("subdevice",          ctypes.POINTER(ctypes.c_uint)),
             ("channel",            ctypes.POINTER(ctypes.c_uint)),
             ("start_counter",      ctypes.POINTER(ctypes.c_ulonglong)),
             ("time_tag",           ctypes.POINTER(ctypes.c_uint)),
             ("dif1",               ctypes.POINTER(ctypes.c_uint)),
             ("dif2",               ctypes.POINTER(ctypes.c_uint)),
             ("time",               ctypes.POINTER(ctypes.c_ulonglong)),
             ("master_rst_counter", ctypes.POINTER(ctypes.c_uint)),
             ("adc",                ctypes.POINTER(ctypes.c_int)),
             ("signal1bit",         ctypes.POINTER(ctypes.c_ushort)),
```

(continues on next page)

(continued from previous page)

```
            ("som_indices_len",     ctypes.c_uint),
            ("ms_indices_len",      ctypes.c_uint),
            ("data_len",            ctypes.c_uint),
            ("reserved",            ctypes.c_char * 12)]
```

3.2 class Device

class scTDC.**Device**(*inifilepath='tdc_gpx3.ini'*, *autoinit=True*, *lib=None*)

A higher-level interface for TDCs and DLDs for applications that use pre-computed histograms from the scTDC library.

__init__(*inifilepath='tdc_gpx3.ini'*, *autoinit=True*, *lib=None*)

Creates a device object.

Parameters

- **inifilepath** (*str, optional*) – the name of or full path to the configuration/ini file, defaults to "tdc_gpx3.ini"
- **autoinit** (*bool, optional*) – if True, initialize the hardware immediately, defaults to True
- **lib** (*scTDClib*, optional) – if not None, reuse the specified scTDClib object, else the Device class creates its own scTDClib object internally, defaults to None

initialize()

Initialize the hardware.

Returns a tuple containing an error code and a human-readable error message (zero and empty string in case of success)

Return type tuple(int, str)

deinitialize()

Deinitialize the hardware.

Returns a tuple containing an error code and a human-readable error message (zero and empty string in case of success)

Return type tuple(int, str)

is_initialized()

Query whether the device is initialized

Returns True if the device is initialized

Return type bool

do_measurement(*time_ms=100*, *synchronous=False*)

Start a measurement.

Parameters

- **time_ms** (*int, optional*) – the measurement time in milliseconds, defaults to 100
- **synchronous** (*bool, optional*) – if True, block until the measurement has finished. defaults to False.

Returns a tuple (0, "") in case of success, or a negative error code and a string with the error message

Return type tuple(int, str)

`interrupt_measurement()`

Interrupt a measurement that was started with synchronous=False.

Returns a tuple (0, "") in case of success, or a negative error code and a string with the error message

Return type tuple(int, str)

`add_end_of_measurement_callback(cb)`

Adds a callback function for the end of measurement. The callback function needs to accept one `int` argument which indicates the reason for the callback. Notification via callback is useful if you want to use do_measurement(...) with synchronous=False, for example in GUIs that need to be responsive during measurement.

Parameters **cb** (*Callable*) – the callback function

Returns non-negative ID of the callback (for later removal)

Return type int

`remove_end_of_measurement_callback(id_of_cb)`

Removes a previously added callback function for the end of measurement.

Parameters **id_of_cb** (*int*) – the ID as previously returned by *add_end_of_measurement_callback*.

Returns 0 on success, -1 if the id_of_cb is unknown

Return type int

`add_3d_pipe(depth, modulo, binning, roi)`

Adds a 3D pipe (x, y, time) with static buffer. The 3D buffer retrieved upon reading is organized such that a point (x, y, time_slice) is addressed by x + y * size_x + time_slice * size_x * size_y. When getting a numpy array view/copy of the buffer, the 'F' (Fortran) indexing order can be chosen, such that the indices are intuitively ordered as x, y, time.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64, BS_FLOAT32, BS_FLOAT64
- **modulo** (*int*) – If 0, no effect. If > 0, a module operation is applied to the time values of events before sorting events into the 3D buffer. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*(int,int,int)*) – a 3-tuple specifying the binning in x, y, and time, where all values need to be a power of 2.
- **roi** (*((int, int), (int, int), (int, int))*) – a 3-tuple of (offset, size) pairs specifying the ranges along the x, y, and time axes.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_xy_pipe(*depth*, *modulo*, *binning*, *roi*)

Adds a 2D pipe (x,y) with static buffer. The 2D buffer retrieved upon reading is organized such that a point (x,y) is addressed by x + y * size_x. When getting a numpy array view/copy of the buffer, the 'F' (Fortran) indexing order can be chosen, such that the indices are intuitively ordered x, y. The binning in time has an influence only on the time units in the roi. The time part in the roi specifies the integration range, such that only events inside this time range are inserted into the data buffer.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64, BS_FLOAT32, BS_FLOAT64
- **modulo** (*int*) – If 0, no effect. If > 0, a module operation is applied to the time values of events before sorting events into the 2D buffer. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*(int,int,int)*) – a 3-tuple specifying the binning in x, y, and time, where all values need to be a power of 2.
- **roi** (*((int, int), (int, int), (int, int))*) – a 3-tuple of (offset, size) pairs specifying the ranges along the x, y, and time axes.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_xt_pipe(*depth*, *modulo*, *binning*, *roi*)

Adds a 2D pipe (x,t) with static buffer. The 2D buffer retrieved upon reading is organized such that a point (x,time) is addressed by x + time * size_x. When getting a numpy array view/copy of the buffer, the 'F' (Fortran) indexing order can be chosen, such that the indices are intuitively ordered x, time. The binning in y has an influence only on the y units in the roi. The y part in the roi specifies the integration range, such that only events inside this y range are inserted into the data buffer.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64, BS_FLOAT32, BS_FLOAT64
- **modulo** (*int*) – If 0, no effect. If > 0, a module operation is applied to the time values of events before sorting events into the 2D buffer. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*(int,int,int)*) – a 3-tuple specifying the binning in x, y, and time, where all values need to be a power of 2.
- **roi** (*((int, int), (int, int), (int, int))*) – a 3-tuple of (offset, size) pairs specifying the ranges along the x, y, and time axes.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_yt_pipe(*depth*, *modulo*, *binning*, *roi*)

Adds a 2D pipe (y,t) with static buffer. The 2D buffer retrieved upon reading is organized such that a point (y,time) is addressed by y + time * size_y. When getting a numpy array view/copy of the buffer, the 'F' (Fortran) indexing order can be chosen, such that the indices are intuitively ordered y, time. The binning in x has an influence only on the x units in the roi. The x part in the roi specifies the integration range, such that only events inside this x range are inserted into the data buffer.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64, BS_FLOAT32, BS_FLOAT64

- **modulo** (*int*) – If 0, no effect. If > 0, a module operation is applied to the time values of events before sorting events into the 2D buffer. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*(int,int,int)*) – a 3-tuple specifying the binning in x, y, and time, where all values need to be a power of 2.
- **roi** (*((int, int), (int, int), (int, int))*) – a 3-tuple of (offset, size) pairs specifying the ranges along the x, y, and time axes.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_t_pipe(*depth*, *modulo*, *binning*, *roi*)

Adds a 1D time histogram pipe, integrated over a rectangular region in the (x,y) plane (for delay-line detectors) with static buffer. The buffer received upon reading is a 1D array of the intensity values for all resolved time bins. The binning in x and y has an influence only on the x and y units in the roi. The x and y parts in the roi specify the integration ranges, such that only events inside the x and y ranges are inserted into the data buffer.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64, BS_FLOAT32, BS_FLOAT64
- **modulo** (*int*) – If 0, no effect. If > 0, a module operation is applied to the time values of events before sorting events into the 1D array. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*(int,int,int)*) – a 3-tuple specifying the binning in x, y, and time, where all values need to be a power of 2.
- **roi** (*((int, int), (int, int), (int, int))*) – a 3-tuple of (offset, size) pairs specifying the ranges along the x, y, and time axes.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_statistics_pipe()

Adds a pipe for statistics data (sometimes referred to as rate meters). The statistics data is only updated at the end of each measurement.

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

add_tdc_histo_pipe(*depth*, *channel*, *modulo*, *binning*, *offset*, *size*)

Adds a pipe for time histograms from a stand-alone TDC. TDC events are filtered by the specified channel, and their time values are transformed first by modulo, then by binning, then by subtraction of the offset, and finally, by clipping to the size value. The resulting value (if not clipped) is the array index of the histogram which is incremented by one.

Parameters

- **depth** (*int*) – one of BS8, BS16, BS32, BS64
- **channel** (*int*) – selects the TDC channel

- **modulo** (*int*) – If 0, no effect. If > 0, a modulo operation is applied to the time values of TDC events before sorting them into the 1D array. The unit of the module value is the time bin divided by 32, i.e. a modulo value of 32 corresponds to one time bin.
- **binning** (*int*) – divides the time value by the specified binning before sorting into the 1D array. Must be a power of 2. Binning 1 is equivalent to no binning.
- **offset** (*int*) – The offset / lower boundary of the accepted range on the time axis.
- **size** (*int*) – The size / length of the accepted range on the time axis. The size is also directly the number of entries in the array/histogram

Returns a tuple containing a non-negative pipe ID and the Pipe object in case of success. A tuple containing the negative error code and the error message in case of failure.

Return type tuple(int, *Pipe*) | tuple(int, str)

remove_pipe(*pipeid*)

Remove a pipe. Manual removal of pipes may be unnecessary if you are using all created pipes until the deinitialization of the device.

Parameters **pipeid** (*int*) – the pipe ID as returned in the first element of the tuple by all add_XYZ_pipe functions

Returns 0 on success, -1 if pipe id unknown or error

Return type int

3.2.1 Pixel/Voxel data type constants

scTDC.**BS8 = 0**

pixel data format, unsigned 8-bit integer

scTDC.**BS16 = 1**

pixel data format, unsigned 16-bit integer

scTDC.**BS32 = 2**

pixel data format, unsigned 32-bit integer

scTDC.**BS64 = 3**

pixel data format, unsigned 64-bit integer

scTDC.**BS_FLOAT32 = 4**

pixel data format, single-precision floating point number

scTDC.**BS_FLOAT64 = 5**

pixel data format, double-precision floating point number

3.3 class Pipe

class scTDC.**Pipe**(*typestr*, *par*, *parent*)

This class handles various types of data received from scTDC library pipes, such as

- 1D, 2D, 3D histograms from DLD (detected particle) events
- statistics data at the end of measurements
- time histograms from stand-alone TDCs.

Instantiation of this class should only happen through calls to the add_XYZ_pipe functions from the *Device* class. Use methods of this class to access the data produced by the Pipe.

__init__(*typestr*, *par*, *parent*)

Constructs a Pipe object. Creates the data buffer and opens the pipe in the scTDC library for the parent Device.

Parameters

- **typestr** (*str*) – one of '3d', 'xy', 'xt', 'yt', 't', 'stat'
- **par** (sc_pipe_dld_image_xyt_params_t | sc_pipe_statistics_params_t) – pipe configuration parameters
- **parent** (*Device*) – a *Device* object

is_open()

Query whether the pipe is active / open.

Returns True, if the pipe is active in the scTDC library (if so, the library writes to the data buffer during measurements and increments histogram entries on incoming events).

Return type bool

reopen(*force=False*)

Open a pipe with previous parameters, if not currently open.

Parameters **force** (*bool, optional*) – set this to True, if the pipe has not been explicitly closed, but the device was deinitialized, causing an implicit destruction of the pipe (implicit desctruction only happens through low-level API calls, whereas Device.deinitialize will close all pipe objects and delete references to them), defaults to False

Returns None if nothing to do, (0, "") on success, (error code, message) on failure

Return type None | (int, str)

close()

Close the pipe such that no events are sorted into the data buffer anymore. The data buffer remains unchanged. In that sense, closing acts more like setting the pipe inactive the pipe can be reopened later. The data buffer can only be garbage-collected after deleting the pipe object via the parent device and discarding all other references to the Pipe object.

Returns (0, "") if success, (error code, message) on failure

Return type (int, str)

get_buffer_view()

For 1D, 2D, 3D pipes, get a numpy array of the data buffer, constructed without copying. As a consequence, changes to the data buffer, made by the scTDC library after getting the buffer view, will be visible to the numpy array returned from this function. The indexing is in Fortran order, i.e. x, y, time. If the pipe is a statistics pipe, get the statistics_t object which may be modified subsequently by the scTDC.

Returns a view of the static data buffer

Return type numpy.ndarray | *statistics_t*

get_buffer_copy()

For 1D, 2D, 3D pipes, get a numpy array of a copy of the data buffer. The indexing is in Fortran order, i.e. x, y, time. If the pipe is a statistics pipe, return a copy of the statistics_t object.

Returns a copy of the data buffer

Return type numpy.ndarray | *statistics_t*

clear()

Set all voxels of the data buffer to zero

3.4 class buffered_data_callbacks_pipe

class scTDC.**buffered_data_callbacks_pipe**(*lib*, *dev_desc*, *data_field_selection=64*, *max_buffered_data_len=65536*, *dld_events=True*)

Base class for using the BUFFERED_DATA_CALLBACKS interface which provides DLD or TDC events in a list-of-events form. Requires scTDC1 library version >= 1.3010.0. In comparison to the USER_CALLBACKS pipe, this pipe reduces the number of callbacks into python, buffering a higher number of events within the library before invoking the callbacks. Thereby, the number of Python lines of code that need to be executed can be drastically reduced if you stick to numpy vector operations rather than iterating through the events one by one. The *on_data* callback receives a dictionary containing 1D numpy arrays where the size of these arrays can be as large as specified by the max_buffered_data_len parameter. To use this interface, write a class that derives from this class and override the methods

- *on_data*
- *on_end_of_meas*

__init__(*lib*, *dev_desc*, *data_field_selection=64*, *max_buffered_data_len=65536*, *dld_events=True*)

Creates the pipe which will be immediately active until closed. Requires an already initialized device.

Parameters

- **lib** (*scTDClib*) – an *scTDClib* object
- **dev_desc** (*int*) – device descriptor as returned by *sc_tdc_init_inifile* or *sc_tdc_init_inifile_overrides*
- **data_field_selection** (*int*) – a 'bitwise or' combination of SC_DATA_FIELD_xyz constants, defaults to *SC_DATA_FIELD_TIME*
- **max_buffered_data_len** (*int*) – The number of events that are buffered before invoking the on_data callback. Less events can also be received in the on_data callback, when the user chooses to return True from the on_end_of_meas callback. defaults to (1<<16)
- **dld_events** (*bool*) – if True, receive DLD events. If False, receive TDC events. Depending on the configuration in the tdc_gpx3.ini file, only one type of events may be available. defaults to True

on_data(*data*)

Override this method to process the data.

Parameters **data** – A dictionary containing several numpy arrays. The selection of arrays depends on the data_field_selection value used during initialization of the class. The following key names are always present in this dictionary:

- event_index
- data_len

Keywords related to regular event data are:

- subdevice
- channel
- start_counter

- time_tag
- dif1
- dif2
- time
- master_rst_counter
- adc
- signal1bit

Keywords related to indexing arrays are:

- som_indices (start of a measurement)
- ms_indices (millisecond tick as tracked by the hardware)

These contain event indices that mark the occurence of what is described in parentheses in the above list.

Returns None

`on_end_of_meas()`

Override this method to trigger actions at the end of the measurement. Do not call methods that start the next measurement from this callback. This cannot succeed. Use a signalling mechanism into your main thread, instead.

Returns True indicates that the pipe should transfer the remaining buffered events immediately after returning from this callback. False indicates that the pipe may continue buffering the next measurements until the max_buffered_data_len threshold is reached.

Return type bool

`close()`

Close the pipe.

`start_measurement_sync(time_ms)`

Start a measurement and wait until it is finished.

Parameters **time_ms** (*int*) – the duration of the measurement in milliseconds.

Returns 0 on success or a negative error code.

Return type int

`start_measurement(time_ms, retries=3)`

Start a measurement 'in the background', i.e. don't wait for it to finish.

Parameters

- **time_ms** (*int*) – the duration of the measurement in milliseconds.
- **retries** (*int*) – in an asynchronous scheme of measurement sequences, trying to start the next measurement can occasionally result in a "NOT READY" error. Often some thread of the scTDC1 library just needs a few more cycles to reach the "idle" state again, where the start of the next measurement will be accepted. The retries parameter specifies how many retries with 0.001 s sleeps in between will be made before giving up, defaults to 3

Returns 0 on success or a negative error code.

Return type int

scTDC.**SC_DATA_FIELD_SUBDEVICE = 1**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_CHANNEL = 2**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_START_COUNTER = 4**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_TIME_TAG = 8**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_DIF1 = 16**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_DIF2 = 32**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_TIME = 64**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_MASTER_RST_COUNTER = 128**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_ADC = 256**

used in buffered_data_callbacks_pipe

scTDC.**SC_DATA_FIELD_SIGNAL1BIT = 512**

used in buffered_data_callbacks_pipe

3.5 class usercallbacks_pipe

class scTDC.**usercallbacks_pipe**(*lib*, *dev_desc*)

Base class for user implementations of the "USER_CALLBACKS" interface. Derive from this class and override some or all of the methods

- on_start_of_meas
- on_end_of_meas
- on_millisecond
- on_statistics
- on_tdc_event
- on_dld_event

The lib argument in the constructor expects a scTDClib object. The dev_desc argument in the constructor expects the device descriptor as returned by sc_tdc_init_inifile(...).

__init__(*lib*, *dev_desc*)

3.6 class Camera

class scTDC.**Camera**(*inifilepath='tdc_gpx3.ini'*, *autoinit=True*, *lib=None*)

Bases: *scTDC.Device*

A specialization of the *Device* class that offers additional, camera-specific functions

add_frame_pipe()

Add a CamFramePipe pipe for receiving meta data and image data for individual camera frames.

Returns a tuple containing a pipe ID and the *CamFramePipe* object if successful; a tuple containing the error code and an error message in case of failure.

Return type (int, *CamFramePipe*) | (int, str)

add_blobs_pipe()

Add a CamBlobsPipe for receiving blob data for individual camera frames.

Returns a tuple containing a pipe ID and the *CamBlobsPipe* object if successful; a tuple containing the error code and an error message in case of failure.

Return type (int, *CamBlobsPipe*) | (int, str)

set_exposure_and_frames(*exposure*, *nrframes*)

Set the exposure per frame in microseconds and the number of frames

Parameters

- **exposure** (*int*) – the exposure per frame in microseconds
- **nrframes** (*int*) – the number of frames

Returns (0, "") on success or a tuple with negative error code and error message

Return type (int, str)

get_max_size()

Get the maximum possible width and height for regions of interest (the width and height in pixels of the sensor area).

Returns a tuple (0, roi) where roi is a dictionary with keys 'width' and 'height' if successful, a tuple (error_code, error_message) in case of failure

Return type (int, dict) | (int, str)

set_region_of_interest(*xmin*, *xmax*, *ymin*, *ymax*)

Set the region of interest.

Parameters

- **xmin** (*int*) – the position of the boundary to the left
- **xmax** (*int*) – the position of the boundary to the right
- **ymin** (*int*) – the position of the top boundary
- **ymax** (*int*) – the position of the bottom boundary

Returns a tuple (0, "") in case of success, or a tuple containing a negative error code and an error message

Return type (int, str)

`get_region_of_interest()`

Get the currently set region of interest

Returns a tuple (0, roi) where roi is a dict with keywords 'xmin', 'xmax', 'ymin', 'ymax' if successful, a tuple containing error code and error message in case of failure

Return type (int, dict) | (int, str)

`set_fanspeed(fanspeed)`

Set the fan speed

Parameters **fanspeed** (*int*) – the fan speed on a scale from 0 (off) to 255 (maximum)

Returns (0, "") if successful; (error_code, error_message) in case of failure

Return type (int, str)

`set_blob_mode(blob_dif_min_top=1, blob_dif_min_bottom=3)`

Activate blob mode. Refer to https://www.surface-concept.com/sctdc-sdk-doc/05_reconflex_cameras.html#blob-recognition-criteria, (condition 3 + 4) for explanation of the blob_dif_min_top/bottom parameters.

Parameters

- **blob_dif_min_top** (*int*) – allowed values range from 0 to 63
- **blob_dif_min_bottom** (*int*) – allowed values range from 0 to 63

Returns (0, "") if successful; (error_code, error_message) in case of failure

Return type (int, str)

`set_image_mode()`

Deactivate blob mode.

`set_smoother_masks_square(size1=1, size2=1)`

Set the smoother pixel masks to filled squares of specified sizes. Specifying both sizes as 1 results in no smoothing. Allowed values for each of the size parameters are 1, 2, 3, 4, 5.

`set_smoother_bit_shifts(shift1=0, shift2=0)`

Set the smoother bit shifts applied to the intensity value after convolution with the smoother pixel mask after smoothing stages 1 and 2, respectively. Recommended shifts for square sizes used in *set_smoother_masks_square* are shift 0 for size 1, shift 2 for size 2, shift 3 for size 3, shift 4 for size 4, shift 4 or 5 for size 5.

`set_analog_gain(value)`

Set the analog gain (a property of the sensor)

Parameters **value** (*int*) – the analog gain value ranging from 0 to 480

Returns (0, "") in case of success; (error_code, error_message) in case of failure

Return type (int, str)

`get_analog_gain()`

Get the currently set analog gain (a property of the sensor)

Returns A tuple (0, analog_gain) if successful; a tuple (error_code, error_message) in case of failure

Return type (int, int) | (int, str)

`set_black_offset`(*value*)

Set the black offset (a property of the sensor)

Parameters `value` (*int*) – the black offset value ranging from 0 to 255 (in BitMode 8) or from 0 to 4095 (in BitMode 12).

Returns (0, "") in case of success; (error_code, error_message) in case of failure

Return type (int, str)

`get_black_offset`()

Get the currently set black offset (a property of the sensor)

Returns A tuple (0, black_offset) if successful; a tuple (error_code, error_message) in case of failure

Return type (int, int) | (int, str)

`set_white_pixel_min`(*value*)

Set the 'White Pixel Min' parameter, a threshold criterion for filtering white pixels.

Parameters `value` (*int*) – the white pixel minimum value ranging from 0 to 255. 0 turns white pixel remover off. 1 is the lowest threshold (removes the most white pixels).

Returns (0, "") in case of success; (error_code, error_message) in case of failure

Return type (int, str)

`get_white_pixel_min`()

Get the current value of the 'White Pixel Min' parameter

Returns A tuple (0, white_pixel_min) if successful; a tuple (error_code, error_message) in case of failure

Return type (int, int) | (int, str)

`set_white_pixel_relax`(*value*)

Set the 'White Pixel Relax' parameter, which controls a ratio between the center pixel and its horizontal and vertical neighbours such that if the ratio is exceeded, the center pixel is considered as a white pixel

Parameters `value` (*int*) – the white pixel relax value, one of 0, 1, 2, 3.

- white pixel relax == 0 : ratio 2;
- white pixel relax == 1 : ratio 1.5;
- white pixel relax == 2 : ratio 1.25;
- white pixel relax == 3 : ratio 1;

Returns (0, "") in case of success; (error_code, error_message) in case of failure

Return type (int, str)

`get_white_pixel_relax`()

Get the current value of the 'White Pixel Relax' parameter

Returns A tuple (0, white_pixel_relax) if successful; a tuple (error_code, error_message) in case of failure

Return type (int, int) | (int, str)

set_shutter_mode(*value*)

Set the shutter mode

Parameters **value** (*int*) – one of the values defined in the *ShutterMode* class

Returns (0, "") in case of success; (error_code, error_message) in case of failure

Return type (int, str)

get_shutter_mode()

Get the currently active shutter mode

Returns A tuple (0, shutter_mode) if successful; a tuple (error_code, error_message) in case of failure. The shutter mode value is one of the constants defined in the *ShutterMode* class

Return type (int, int) | (int, str)

class scTDC.**ShutterMode**

Defines the values that represent the available shutter modes

START_AND_STOP_BY_WIRE = 0

START_AND_STOP_BY_SOFTWARE = 1

START_BY_WIRE_STOP_BY_SOFTWARE = 2

START_BY_SOFTWARE_STOP_BY_WIRE = 3

3.7 class CamFramePipe

class scTDC.**CamFramePipe**(*device*)

A pipe for reading camera image frames and frame meta information synchronously. Do not instantiate this class by hand. Use Camera.add_frame_pipe, instead.

is_active()

Query whether the pipe is active / open.

Returns True if the pipe is active.

Return type bool

close()

Close the pipe, release memory associated with the pipe

read(*timeout_ms=500*)

Wait until the next camera frame becomes available or timeout is reached. Return meta data and, if available, image data of the next camera frame. If image data is returned, access to it is only allowed until the next time that this read function is called. Perform a copy of the image data if you need to keep it for longer. As soon as a CamFramePipe is opened and measurements are started, the pipe allocates memory for storing the frame data until this frame data is read. Not reading the pipe frequently enough can exhaust the memory.

Parameters **timeout_ms** (*int, optional*) – the timeout in milliseconds, defaults to 500

Returns Returns a tuple (meta, image_data) where meta is a dictionary containing the frame meta data, and image_data is a numpy array. If an error occurs, returns a tuple (error_code, error_message).

Return type (dict, numpy.ndarray) | (int, str)

3.8 class CamBlobsPipe

class scTDC.**CamBlobsPipe**(*device*)

A pipe for reading camera blob data synchronously. Do not instantiate this class by hand. Use Camera.add_blobs_pipe, instead.

is_active()

Query whether the pipe is active / open.

Returns True if the pipe is active.

Return type bool

close()

Close the pipe, release memory associated with the pipe

read(*timeout_ms=500*)

Wait until blob data for the next camera frame becomes available or timeout is reached and read it. Returns blob data of the next camera frame. If data is returned, access is only allowed until the next time that this read function is called. Perform a copy of the data if you need to keep it for longer. As soon as a CamBlobsPipe is opened and measurements are started, the pipe allocates memory for storing data which is released by reading. Not reading the pipe frequently enough can exhaust the memory.

Parameters **timeout_ms** (*int, optional*) – the timeout in milliseconds, defaults to 500

Returns Returns an array of blob positions in case of success. If an error occurs, returns a tuple (error_code, error_message).

Return type numpy.ndarray | (int, str)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

G

I

O

P