scTDC SDK

Release 1.3019.8

Surface Concept GmbH

Nov 15, 2023

Contents:

1 Introduction **3**

- 1.1 Scope . . . 3
- 1.2 SDK contents . . . 3
- 1.3 Alternatives . . . 3
- 1.4 Features . . . 4
- 1.5 General usage scheme . . . 4

2 Initialization of a device **5**

- 2.1 Device descriptor . . . 5
- 2.2 Error reporting . . . 5
- 2.3 Initialization errors . . . 5
- 2.4 Requirements to the placement of files . . . 6
- 2.5 Overriding settings of the ini file by software . . . 6
  - 2.5.1 `sc_ConfigLine` interface . . . 6
  - 2.5.2 “override registry” interface . . . 7

3 Starting a measurement **9**

- 3.1 Working with hardware-triggered measurements . . . 9
- 3.2 Working with external start pulses (for time-resolved measurements) . . . 9
- 3.3 Waiting for the end of a measurement . . . 10
- 3.4 Running sequences of measurements . . . 11

4 Receiving Data **13**

- 4.1 Common configuration parameters and concepts . . . 13
  - 4.1.1 TDC events and DLD events . . . 13
  - 4.1.2 time value of TDC and DLD events . . . 14
  - 4.1.3 configuration parameter `depth` . . . 14
  - 4.1.4 configuration parameter `binning` and region of interest (`roi`) . . . 14
  - 4.1.5 configuration parameter `modulo` . . . 15
  - 4.1.6 Memory handling for pipe data (allocator_owner, allocator_cb parameters) . . . 15
- 4.2 Overview of the various types of data pipes . . . 15
  - 4.2.1 TDC_HISTO . . . 16
  - 4.2.2 DLD_IMAGE_XY . . . 16
  - 4.2.3 DLD_IMAGE_XT . . . 16
  - 4.2.4 DLD_IMAGE_YT . . . 17
  - 4.2.5 DLD_IMAGE_3D . . . 17
  - 4.2.6 DLD_SUM_HISTO . . . 17
  - 4.2.7 STATISTICS . . . 17
  - 4.2.8 TMSTAMP_TDC_HISTO, TDC_STATISTICS, DLD_STATISTICS . . . 18
  - 4.2.9 USER_CALLBACKS . . . 18

4.2.10 DLD_IMAGE_XY_EXT . . . 18
4.2.11 BUFFERED_DATA_CALLBACKS . . . 18
4.2.12 PIPE_CAM_FRAMES . . . 18
4.2.13 PIPE_CAM_BLOBS . . . 19
4.3 User callbacks interface . . . 19
4.3.1 DLD applications . . . 19
4.3.2 TDC applications . . . 20

5 Recon*Flex*™ cameras **23**
5.1 Configuring camera parameters . . . 23
5.1.1 Setting exposure (per frame) and number of frames for measurements . . . 23
5.1.2 Selecting *raw image* mode or *blob* mode . . . 23
5.1.3 Blob recognition criteria . . . 24
5.1.4 Setting a region of interest (ROI) on the hardware level . . . 25
5.1.5 Selecting the dynamics range of intensities in raw images . . . 25
5.2 Receiving Data . . . 26
5.2.1 DLD_IMAGE_XY pipe versus PIPE_CAM_FRAMES pipe . . . 26
5.2.2 User callbacks interface versus PIPE_CAM_BLOBS pipe . . . 26
5.2.3 Combining PIPE_CAM_FRAMES and PIPE_CAM_BLOBS pipe . . . 27

6 Code Examples **29**
6.1 Time Histogram Pipe (stand-alone TDC) . . . 29
6.2 Image Pipe . . . 30
6.3 External Memory Allocation . . . 32
6.4 User Callbacks Pipe . . . 34
6.5 Camera pipes for frame meta info, raw images and blobs . . . 36

7 Old API notes **41**

8 scTDC library API **43**
8.1 Page Hierarchy . . . 43
8.2 Class Hierarchy . . . 43
8.3 File Hierarchy . . . 43
8.4 Full API . . . 43
8.4.1 Classes and Structs . . . 43
8.4.2 Enums . . . 70
8.4.3 Functions . . . 75
8.4.4 Variables . . . 95
8.4.5 Defines . . . 96

Index **113**

The scTDC library is used for developing applications for various products built by Surface Concept GmbH, such as Time-to-Digital Converters (TDC), Delay Line Detector (DLD) devices, and cameras (ReconFlex^{TM} series).

1 Introduction

1.1 Scope

The scTDC software development kit (SDK) for the C and C++ programming languages enables the creation of custom user applications for a selection of products built by Surface Concept GmbH:

- stand-alone time-digital converters (TDC)
- delayline detectors (DLD)
- ReconFlex^{TM} cameras

1.2 SDK contents

This SDK comprises

1. a set of shared libraries (the main library is a file `scTDC1.dll` for MS Windows, or `libscTDC.so` for Linux systems). The library files are provided in two variants:
   - for 32-bit applications (`lib_x86` folder)
   - for 64-bit applications (`lib_x64` folder)
2. header files for C/C++. User code needs to `#include <...>` these files to use functions of the library
3. code examples (in separate files + *in this document*)
4. this document

1.3 Alternatives

Custom user applications can also be developed using

1. our Python SDK
2. our LabView instrument driver.

These alternatives make it easier to learn about application development for our products and may provide better integration with existing user software (for example, integration with other devices at the user site which are already controlled from LabView).

1.4 Features

The strong points of this SDK for C/C++ are

1. access to the highest possible performance
2. highest flexibility with regard to reading and processing of data
3. full coverage of functionality. This SDK will always provide the newest features first, before they will be integrated into the Python and LabView support.

1.5 General usage scheme

Any application for operating the products supported by this SDK will involve the following steps:

1. Initialization of the hardware (TDC, DLD, or camera)
2. Configuring which kind of data are produced by the library
3. Starting a measurement (once or repeatedly)
4. Reading and processing of data
5. Deinitialization of the hardware

These steps will be covered in the following sections.

2 Initialization of a device

To initialize (or "configure") a device, the application must point the library to a configuration file, shipped as part of the product. This configuration file is typically named "tdc_gpx3.ini" and frequently referred to as the *ini file*. The *ini file* may contain references to other files (such as firmware files, calibration data and others). These files are best stored in the same folder as the ini file.

Initialization is done by calling `sc_tdc_init_inifile()` with the name of the ini file or the full path to that file.

2.1 Device descriptor

A call to `sc_tdc_init_inifile()` returns a non-negative (>= 0) integer if the initialization was successful. This number is referred to as a **device descriptor** in the API of the library and must be used in all other function calls related to the initialized device.

2.2 Error reporting

If initialization was not successful, a negative value – an error code – is returned, indicating the reason of failure. Error codes are listed in file_scTDC_error_codes.h and their macro names start with `SC_TDC_ERR_`. A human-readable, textual description of the error can be obtained using the `sc_get_err_msg()` function. Most other functions of the scTDC library, if they return a signed integer type, indicate errors by returning negative values that translate into textual descriptions via `sc_get_err_msg()`.

2.3 Initialization errors

Common error codes returned from `sc_tdc_init_inifile()` and its variants are

- `-9` (`SC_TDC_ERR_BADCONFI`): either the specified *ini file* was not found, or it contained syntactic errors.
- `-12` (`SC_TDC_ERR_DEVCLS_LD`): *secondary libraries* were missing or did not have the right variant between 32-bit and 64-bit (see *Requirements to the placement of files*)
- `-15` (`SC_TDC_ERR_FPGA_INIT`): something went wrong inside the hardware while trying to initialize it. Resetting the hardware with a power cycle (power off + power on) may help. Providing the wrong firmware file can also lead to this error code.
- `-98` (`SC_TDC_ERR_NO_DEVICE`): no device was found. Is the device powered on and connected to the PC? Often there is a way to check hardware connectivity from the operating system (such as in the device manager of Windows for USB devices, or the Ethernet adapter dialog of Windows for Ethernet-based devices).

2.4 Requirements to the placement of files

This section is written with Windows as the operating system in mind:

An application that is linked to the scTDC library can only be started if the library file `scTDC1.dll` and a second dependent library `pthreadVC2.dll` are found, which can be ensured by putting these files in the same folder as the application executable (`*.exe`) file.

At the point where the application initializes the device, the scTDC library attempts to dynamically load *secondary libraries*, and read configuration files which should also be put into the folder of the application executable. In case of libraries, make sure to pick the right variant between 32-bit and 64-bit. The following is a list of files whose presence is required for the initialization step:

- 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 with a `scDeviceClass60.dll` require a firmware file with the `*.bit` file name extension.
- if your demo software folder includes files `cal_i.tif`, `cal_xyt.txt`, they should be copied to the folder of your application.

Missing *secondary libraries* result in an error code `-12` (`SC_TDC_ERR_DEVCLS_LD`) returned by the initialization function.

2.5 Overriding settings of the ini file by software

Of the parameters listed in the ini file, some can be changed after initializing the device, whereas others have to be set before initialization and cannot be changed afterwards. For the latter type, there are means to modify the parameter values by application code before initializing the device while leaving the original ini file untouched.

There are two options for overriding settings of the ini file. The *sc_ConfigLine* interface was the first available option. The newer “override registry” interface is recommended for development of newer applications.

2.5.1 `sc_ConfigLine` interface

This interface requires application code to provide **all entries** of the ini file in an array of *sc_ConfigLine* structs. This may make it necessary for the application code to parse the ini file. After initialization of the device, there is no connection to any *ini file* on hard disk which prevents live-tuning of certain ini file parameters during sequences of measurements in the `DebugLevel > 0` modes — sometimes used for diagnosis and calibration.

Prepare an array of *sc_ConfigLine* structs, then pass it into *sc_tdc_init_with_config_lines()* to initialize the device.

2.5.2 "override registry" interface

This interface enables initialization of a device from an *ini file*, as usual, while replacing only a selected set of parameters with modified values. After initialization, a connection to the *ini file* on hard disk remains and live-tuning of ini file parameters in the `DebugLevel > 0` modes is possible.

Call `sc_tdc_overrides_create()` to prepare a new *override registry*, then add entries to it using `sc_tdc_overrides_add_entry()`. Finally, call `sc_tdc_init_inifile_override()` to initialize the device. The library copies the contents and associates them with the device descriptor. After initialization, it is possible to close the *override registry* via `sc_tdc_overrides_close()`, to release its memory, while the overridden values remain in effect for the initialized device.

3 Starting a measurement

The function *sc_tdc_start_measure2()* is used to start a measurement. After calling this function, the device switches into a measurement state. The library receives and processes data from the device in a separate thread, and transfers the results to the application through the data pipes that have been configured in advance. See *Receiving Data* for more info on how to operate data pipes.

3.1 Working with hardware-triggered measurements

By default, starting of a measurement comes with a certain variable delay, before the hardware actually enters the state of detecting and recording of data. This default behaviour corresponds to the ini file parameter `ext_trigger` being set to `NO`.

When the ini file parameter `ext_trigger` is set to `YES`, the hardware waits for an external trigger at the *Sync In* input each time the application calls the *sc_tdc_start_measure2()* function — so this function can be called in advance. As soon as the pulse on the *Sync In* input is received by the hardware, it starts the measurement with a duration as specified in the respective argument passed *sc_tdc_start_measure2()*. This is useful for precisely synchronizing the starts of measurements with external devices.

Caution: If no pulse occurs on the *Sync In* input, the hardware remains in a waiting state and does not start the measurement. From the software side, the device is already regarded as being in a measurement state, but the end of the measurement is never reached.

3.2 Working with external start pulses (for time-resolved measurements)

(This subsection applies to TDCs and DLDs but not to cameras). Hardware can be configured to work with external *Start pulses* by setting the ini file parameter `Ext_Gpx_Start` to `YES`. The start of the measurement is not influenced by this setting — by default, the hardware starts the measurement with a short delay after calling *sc_tdc_start_measure2()* and terminates the measurement after the specified exposure. However, for data to be recorded, at least one pulse has to be fed into the *Start In* hardware input. If no such pulse occurs, no data is recorded, such that histogram pipes return empty histograms and USER_CALLBACKS pipes do not deliver any DLD or TDC events.

3.3 Waiting for the end of a measurement

The `sc_tdc_start_measure2()` function starts a measurement in the background and control is immediately returned back to the user code, before the measurement finishes. There are two general options and two use-case specific options to wait for the end of a measurement.

General options:

1. **Polling the idle status** The application developer can repeatedly call `sc_tdc_get_status2()` to ask whether the hardware is still in a measurement. This includes the internal PC-side processing of the scTDC library. The idle state is only reported when the library has finished processing the data stream of the hardware up to the end of the most recently started measurement.

   Between subsequent calls to `sc_tdc_get_status2()`, one typically uses sleep intervals so as to keep the CPU load low. The length of the sleep intervals is chosen as a compromise between fast reaction time and low CPU load. The more frequent you poll, the faster your reaction time for the end of the measurement is. In this sense, polling is always a somewhat inefficient pattern, but it is also easiest to implement. For better performance, read about the next option.

2. **Registering an end-of-measurement callback** Prior to starting a measurement, the application developer can register a function of his choice that the scTDC library calls when the end of measurement is reached. This registration is done by calling `sc_tdc_set_complete_callback2()` and providing a function with the signature `void cb(void* privateptr, int reason_code);`. The proper end of measurement is indicated, when this function is called with a `reason_code == 1`. However, as soon as the hardware stops its exposure, even before all of its data has been transferred to the PC and processed by the library, the `cb` function may be called with a `reason_code == 4`. This code 4 comes in addition and prior to the code 1, so the application developer has to handle this case with a separate behaviour – for example ignoring code 4 and reacting to code 1. Aborted measurements are indicated by `cb` invocations with `reason_code == 2`, and `reason_code == 3`. The reason codes have macro definitions in file_scTDC_error_codes.h (`SC_TDC_INFO_MEAS_COMPLETE`, `SC_TDC_INFO_USER_INTERRUPT`, `SC_TDC_INFO_BUFFER_FULL`, `SC_TDC_INFO_HW_IDLE`)

Use-case specific options:

3. **Reading histogram data or reading statistics data** If your application has configured a pipe that generates histograms or the statistics pipe, you will use `sc_pipe_read2()` to read the respective data. This function can be called with the timeout set to something much longer than the measurement time. As soon as the end of the measurement is reached, histogram data and statistics data become available and the function returns at this moment. This can be used to trigger any actions the application should perform with the data and also to trigger the start of the next measurement.

4. **Reading event data from the *USER_CALLBACKS* pipe and reacting to its *end_of_measure* callback** If your application reads event data via the *USER_CALLBACKS* pipe, one of the callback functions you can register for, notifies you when the end-of-measurement control sign is encountered by the internal hardware protocol decoder of the scTDC library. This control sign appears as the last thing of a measurement, so all measurement data and statistics data have already been processed by the library. However, some internal threads of the library may not have switched to the idle state. It is not possible to start a new measurement directly from the callback function. Instead, you can use thread synchronization mechanisms such as *semaphores* or *condition variables* to notify your main thread, that all data is available and possibly a new measurement can be started.

3.4 Running sequences of measurements

After waiting for the end of a measurement, you may decide to start the next measurement, as soon as possible, thus establishing a sequence of measurements. Depending on how you waited for the end of the measurement you may run into a `SC_TDC_ERR_NOTRDY` (file_scTDC_error_codes.h) returned by calling `sc_tdc_start_measure2()`. This does not happen if you used the polling-for-idle-status mechanism, but it can happen with the other options discussed in the previous section. To make your sequence more robust, you can call `sc_tdc_start_measure2()` multiple times with a thread-yield (meaning your thread pauses for a minimal time until it is scheduled again by the operating system) in between until this function returns zero, indicating success, or some other error than `SC_TDC_ERR_NOTRDY` (in that case you should stop retrying).

4 Receiving Data

Data can be received from the hardware by configuring one or more data pipes before starting measurements. An application can open as many data pipes as required for operation. All of them can have their own parameters, settings and types. The only limit is posed by the machine resources such as memory and processor power. The processing of data pipes is not parallelized. Therefore the amount of CPU cycles required to process one data unit (such as TDC or DLD events, camera blobs, camera raw images) is growing with number of data pipes opened.

Data pipes can be created by calling `sc_pipe_open2()` and read by calling `sc_pipe_read2()`. An exception is the `USER_CALLBACKS` pipe. This pipe does not support reading. During creation of this pipe, the application developer provides a set of callback functions, and the library invokes these functions at the moment where data is produced, or when certain events happen.

`sc_pipe_close2()` is used to close pipes and free associated resources.

4.1 Common configuration parameters and concepts

Before documenting the various pipe types, a few commonly used concepts and configuration parameters are discussed:

4.1.1 TDC events and DLD events

TDCs for applications without a detector (**stand-alone TDCs**) measure an electronic pulse (*Stop pulse*) occurring on one of their *Stop channel* inputs and generate a TDC event for each detected pulse. In the simplest case, this TDC event is only characterized by the index of the *Stop channel* and the digitized time value that reports the time difference between the *Stop pulse* and the *Start pulse*. Furthermore, the TDC event can carry the information of the *Start counter* at the time the event was registered. The *Start counter* keeps track of the number of *Start pulses*. For advanced use cases, additional information can be associated with every TDC event.

TDCs for **delay-line detector** applications need to observe the pulses occurring on multiple channels simultaneously, where each channel corresponds to one of the delay-line terminals. If coincident pulses are found on all channels, the firmware of the TDCs computes a single DLD event and passes it to the PC side (alternatively, the individual channel pulses may be passed to the PC side, so the software can both deliver the TDC events of the individual channels, as well as the combined DLD event). The DLD event carries a time stamp summed up from the TDC events. Additionally the detector position (x,y) has been reconstructed from time differences of the TDC events and is delivered as part of the DLD event. Again, for advanced use cases, additional information can be associated with every DLD event.

4.1.2 time value of TDC and DLD events

Many pipe types involve configuration with regard to a *time* axis. As mentioned in the previous section, TDC events carry a time stamp which is the time difference between Start pulse and Stop pulse (plus an arbitrary constant offset which can be counter-acted by tuning the `CommonShift` ini file parameter). DLD events are combined from multiple coincident TDC events and their time value is the sum of the TDC event time values. The time value is only useful, when the TDC is configured for an external Start pulse (`Ext_Gpx_Start = YES` in the ini file). The opposite setting (`Ext_Gpx_Start = NO`) has the TDC generating internal starts, which are not synchronous to the detected pulses or particles. In this case, the time values are random.

4.1.3 configuration parameter `depth`

This configuration parameter applies to image pipes, 1D histogram pipes and the 3D histogram pipe. These pipes commonly store numbers of detected events which can also be regarded as intensity values. The `depth` parameter controls the data type that is used for these intensity values. Each data type has a different size in memory. There are four data types for unsigned integer numbers of varying size and two data types for floating-point numbers of varying precision. Larger sized data types enable handling of larger intensity values. For example, setting `depth = BS8` means the histogram can only store intensity values up to 255, whereas `depth = BS32` enables a generous maximum intensity above 4.2 x 10^9. Floating-point numbers are useful, if the raw number of detected events shall be corrected by a calibration image. The calibration image introduces floating-point weightings that are within some interval around 1.0.

The depth parameter must be set to one of the enumeration values defined in `bitsize_t`.

4.1.4 configuration parameter `binning` and region of interest (`roi`)

Binning and ROI parameters appear in many pipe types. Binning has the subfields `binning.x`, `binning.y`, `binning.time`. ROI has the subfields `roi.offset.x`, `roi.offset.y`, `roi.offset.time`, `roi.size.x`, `roi.size.y`, `roi.size.time`.

We discuss the effect of these parameters by considering a DLD event, which contains a detector position *(x,y)* and a time stamp *t*. For a particular type and configuration of a histogram pipe, the *(x,y,t)* values can either be within a region of interest or outside. In the latter case, the DLD event is not added to the histogram. In the former case, the correct histogram element must be located and increased by one. To this end, a sequence of computational steps is applied to the DLD event:

- The original *x*, *y*, *t* values are divided by the respective binning values, resulting in x_1, y_1, t_1
- the values specified in `roi.offset` are subtracted from x_1, y_1, t_1, respectively, yielding x_2, y_2, t_2. If any of the values x_2, y_2, t_2 are below zero, the event was outside of the ROI and is discarded.
- x_2, y_2, t_2 are checked to be smaller than the values specified in the respective `roi.size` fields. If any of them do not fulfill this condition, the event was outside of the ROI and is discarded.
- x_2, y_2, t_2 serve as indices into the respective axes of the histogram (for those axes that the histogram contains) to the effect that one histogram element is selected. This element is increased by one. For example, an image pipe of type XY, has two axes, one for detector coordinate x and one for detector coordinate y. The intensity at image position (x_2, y_2) is increased by one, whereas the t_2 has no significance at this point. t_2 was only computed to check the region-of-interest condition for the time, thereby restricting the DLD events that formed the histogram to those that were inside a specified time interval.

This computational sequence defines some conventions how ROI and binning are interpreted.

- the number of histogram elements is entirely defined by the subfields of `roi.size`. The memory requirement follows the number of histogram elements while additionally factoring in the size of one element according to the `depth` parameter.
- The application of binning prior to the application of ROI means that the user-specified ROI is interpreted as given in binned units. Thus, changing the binning from 1 to 2, while leaving the roi parameters unchanged would have the effect that the histogram maps a larger detector area with half the resolution, keeping the number of histogram elements unchanged.

The above description also makes clear, why 1D histograms and 2D histograms for DLD data always use a region-of-interest definition for three axes (x, y, and time). Some of these axes are mapped onto histogram axes, the remaining ones just act as a filter.

Please note, valid values for the binning are powers of 2 (1, 2, 4, 8, 16, 32, …).

4.1.5 configuration parameter `modulo`

Modulo is an additional preprocessing step applied to the time value for those situations where DLD events are periodically generated with a repetition frequency that exceeds the capability of the *Start In* input (start frequencies above 5 MHz are problematic). In that case, a frequency divider is inserted between the original source of the Start pulses and the *Start In* input of the TDC. On the data, the frequency divider has the effect of extending the time range over a multiple of the original time period, showing multiple concatenated replicas of the original data along the time axis. The modulo operation enables adding up and eliminating the replicas, reducing the time range to the original period.

The modulo parameter is configured to the number of time bins corresponding to the time period of your original Start pulse frequency. However, for increased accuracy the modulo parameter is interpreted as a fixed-point decimal number with 5 bits after the comma. Effectively, if you have determined the number of time bins corresponding to your start period, you have to multiply this value by 32 and set this as the modulo value. Fine tuning of the modulo value can be done by repeating measurements and optimizing for sharpness of features in the data. Querying the length of one time bin in nanoseconds is possible via `sc_tdc_get_binsize2()`.

During computation of histograms, the modulo operation is performed before binning and ROI filtering is applied.

4.1.6 Memory handling for pipe data (allocator_owner, allocator_cb parameters)

Due to the non-automatic memory handling in the C programming language, the question of allocation and deallocation of memory is very important. Currently, the scTDC1 has two ways of providing memory buffers for data pipes:

(1) 'internal' memory allocation: Memory buffers are allocated by the scTDC1 in a sequence-of-buffers fashion, such that typically every measurement produces one buffer per pipe (for those pipe types that generate 1D histograms, images, and 3D histograms): When reading such pipes, the first call to sc_pipe_read2() returns the first buffer whereas the second call to sc_pipe_read2() deallocates the first buffer and returns the second buffer. Calling either sc_pipe_close2() or sc_tdc_deinit2() deallocates all remaining buffers.

(2) 'external' memory allocation: The application supplies an allocator callback function in the parameter structure when creating a data pipe. In this case, the allocator callback function is invoked every time the data pipe processing algorithm needs a new buffer for the data (this will typically occur at the start of a measurement). Allocation and deallocation of the memory buffer is then the responsibility of the application. The 'external' memory allocation provides more flexibility to the application developer. For example, the application can choose to reserve a single memory buffer and pass it back each time its allocator callback function is invoked.

For an example demonstrating external memory allocation, see *External Memory Allocation*.

4.2 Overview of the various types of data pipes

The different kinds of pipes that can be opened, are listed in the enum `sc_pipe_type_t`, defined in file_scTDC_types.h.

4.2.1 TDC_HISTO

For stand-alone TDC applications. Provides a histogram that resolves number of detected events as a function of the *time value* on a specified *Stop channel* input. The unit of the time value can be queried via `sc_tdc_get_binsize2()`.

Create a variable of type `sc_pipe_tdc_histo_params_t`, for specifying the configuration of the pipe, then pass its address as the third argument to `sc_pipe_open2()`, whereas the second argument is set to `TDC_HISTO`.

The `channel` field selects the TDC *Stop channel* input.

Binning and a region of interest can be specified for the time values. Allowed binning values are powers of 2 (1, 2, 4, 8, 16, 32, . . .). The fields `offset` and `size` define the lower-end margin and the length of the time interval ROI, where TDC events are only added to the histogram if they are within the ROI. The `size` field also directly defines the number of histogram elements. ROI filtering works similary to what is described in *configuration parameter binning and region of interest (roi)*, except there are no x, y values, just a time value.

Notes concerning configuration parameters:

- See *configuration parameter depth*.
- Set the `modulo` field to 0 or refer to *configuration parameter modulo*.
- See *Memory handling for pipe data (allocator_owner, allocator_cb parameters)*.

4.2.2 DLD_IMAGE_XY

For DLD and camera applications. Provides an image that maps the detector area or the camera sensor area, or just a partial region of those areas.

Create a variable of type `sc_pipe_dld_image_xy_params_t`, fill its fields according to your preferred configuration, then pass its address to `sc_pipe_open2()`, whereas the second argument is set to `DLD_IMAGE_XY`.

Notes concerning configuration parameters:

- See *configuration parameter depth*.
- set the `channel` field to `-1` by default unless your application deals with special hardware types that make use of the channel.
- Set the `modulo` field to 0 or refer to *configuration parameter modulo*.
- See *configuration parameter binning and region of interest (roi)*.
- See *Memory handling for pipe data (allocator_owner, allocator_cb parameters)*.

For a code example, see *Image Pipe*.

4.2.3 DLD_IMAGE_XT

For DLD applications. Provides an image where the image horizontal axis maps the detector horizontal axis, but the image vertical axis maps the time axis of DLD events.

Create a variable of type `sc_pipe_dld_image_xt_params_t`, fill its fields according to your preferred configuration, then pass its address to `sc_pipe_open2()`, whereas the second argument is set to `DLD_IMAGE_XT`.

For details on configuration parameters, see *DLD_IMAGE_XY*.

4.2.4 DLD_IMAGE_YT

For DLD applications. Provides an image where the image horizontal axis maps the detector vertical axis and the image vertical axis maps the time axis of DLD events.

Create a variable of type *sc_pipe_dld_image_yt_params_t*, fill its fields according to your preferred configuration, then pass its address to *sc_pipe_open2()*, whereas the second argument is set to `DLD_IMAGE_YT`.

For details on configuration parameters, see *DLD_IMAGE_XY*.

4.2.5 DLD_IMAGE_3D

For DLD applications. Provides a 3D intensity matrix where the three matrix indices correspond to (1) detector coordinate x, (2) detector coordinate y, (3) time value of the DLD events. The memory requirement for one such matrix can easily reach several Gigabytes, so the `roi.size` parameters in the configuration need to be chosen carefully.

Create a variable of type *sc_pipe_dld_image_3d_params_t*, fill its fields according to your preferred configuration, then pass its address to *sc_pipe_open2()*, whereas the second argument is set to `DLD_IMAGE_3D`.

For details on configuration parameters, see *DLD_IMAGE_XY*.

4.2.6 DLD_SUM_HISTO

For DLD applications. Provides a 1D histogram showing number of events resolved by time value of the DLD event.

Create a variable of type *sc_pipe_dld_sum_histo_params_t*, fill its fields according to your preferred configuration, then pass its address to *sc_pipe_open2()*, whereas the second argument is set to `DLD_SUM_HISTO`.

For details on configuration parameters, see *DLD_IMAGE_XY*.

4.2.7 STATISTICS

For TDC and DLD applications. Provides a statistics record at the end of every measurement. The statistics record is defined by *statistics_t*.

For DLD applications, the statistics record delivers useful diagnostic information to judge the correct calibration of the detector read-out electronics and the particle load on micro-channel plates. Typically, for this purpose, the *statistics_t* fields `counts_read[0][0]` up to `counts_read[0][15]` and the `events_found[0]` as well as the `events_received[0]` are sufficient and should be displayed side by side in one bar graph, if the application has a graphical interface.

Create a variable of type *sc_pipe_statistics_params_t*, fill its fields according to your preferred configuration, then pass its address to *sc_pipe_open2()*, whereas the second argument is set to `STATISTICS`.

Notes concerning configuration parameters:

- See *Memory handling for pipe data (allocator_owner, allocator_cb parameters)*.

4.2.8 TMSTAMP_TDC_HISTO, TDC_STATISTICS, DLD_STATISTICS

(Rarely used and currently undocumented).

4.2.9 USER_CALLBACKS

See *User callbacks interface*.

4.2.10 DLD_IMAGE_XY_EXT

A variant of the DLD_IMAGE_XY pipe which enables an extended set of configuration parameters.

Configuration parameters are defined in `sc_pipe_dld_image_xy_ext_params_t`.

Currently undocumented.

4.2.11 BUFFERED_DATA_CALLBACKS

Used by our implementation of the Python software development kit (SDK). A realization of event data transfer different from the USER_CALLBACKS pipe with the option to largely reduce the frequency of callbacks. The less-frequent callbacks make it better suited for building interfaces to less-performant programming languages.

Configuration parameters are defined in `sc_pipe_buf_callbacks_params_t`.

4.2.12 PIPE_CAM_FRAMES

For camera applications. Provides one data set per camera frame (unlike many other pipes which provide one data set per measurement). The data set includes some frame-related meta information and the raw image data if the camera is currently set up to deliver raw images. This pipe is the most efficient way to access raw image data from the camera since the generation of copies is eliminated as much as possible. The memory buffers transmitted through this pipe are prepared independent of whether a PIPE_CAM_FRAMES pipe is open. In case, multiple PIPE_CAM_FRAMES are open, the memory buffers are shared between these pipes and every pipe instance can read at their own pace. Only those memory buffers that have been read by all PIPE_CAM_FRAMES pipes are being released.

There are zero configuration options for this pipe. Therefore, the third argument to be passed to `sc_pipe_open2()` is `NULL` (C), or `nullptr` (C++).

The data set delivered from `sc_pipe_read2()` is a memory buffer that starts with a `sc_cam_frame_meta_t` structure. The memory buffer remains valid until the next call to `sc_pipe_read2()`. If the `flags` field in this structure has the `SC_CAM_FRAME_HAS_IMAGE_DATA` bit set, the memory buffer contains a second region, which can be reached by advancing the pointer to the memory buffer by the number of bytes given in the field `data_offset`. The size of this second region in bytes can be calculated by `width` times `height` times [1 or 2]. The last factor is 1 if the field `pixelformat` equals `SC_CAM_PIXELFORMAT_UINT8`, or 2 if the field `pixelformat` equals `SC_CAM_PIXELFORMAT_UINT16` (these constants are defined in `sc_cam_pixelformat_t`).

See *Camera pipes for frame meta info, raw images and blobs* for example code.

4.2.13 PIPE_CAM_BLOBS

For camera applications. Provides one data set per camera frame (unlike many other pipes which provide one data set per measurement). The data set comprises a list of blob coordinates pertaining to one camera frame. It can be read in a synchronized fashion with the PIPE_CAM_FRAMES pipe (even in blob mode, if the camera does not deliver raw image data, the PIPE_CAM_FRAMES pipe delivers meta data associated with each camera frame).

There are zero configuration options for this pipe. Therefore, the third argument to be passed to *sc_pipe_open2()* is `NULL` (C), or `nullptr` (C++).

The data set delivered from *sc_pipe_read2()* is a memory buffer that starts with a *sc_cam_blob_meta_t* structure, providing a `data_offset` field and the number of blobs given in the second region of the memory buffer. The second region is reached by advancing the pointer to the memory buffer by as many bytes as given in the `data_offset` field. The content of the second region is an array of element type *sc_cam_blob_position_t* and with a length given by the `nr_blobs` field.

The memory buffer remains valid until the next call to *sc_pipe_read2()*.

See *Camera pipes for frame meta info, raw images and blobs* for example code.

4.3 User callbacks interface

Since version 1.3000.0, a new interface was added for receiving data in a sequence-of-events form which enables a lower-level, more flexible way of processing the data for the application than receiving pre-computed histograms. While such "event-based" pipes already existed before, they required decoding and delicate bit manipulation by the application developer, and they were rather limited in the size of data that could be transported per event. The new interface involves calling *sc_pipe_open2()* with the *sc_pipe_type_t* argument being set to `USER_CALLBACKS` (defined in file_scTDC_types.h). It uses a set of callback functions provided by the application developer so that the library can actively push various kinds of events and data to the application in the correct chronological order as they appear while the library decodes the protocol stream coming from the hardware.

The set of callback functions provided by the application developer is defined in *sc_pipe_callbacks* and comprises callbacks for reacting to

- the start of a measurement
- the end of a measurement
- a millisecond tick recorded by hardware and placed as a marker between TDC or DLD events
- statistics data at the end of a measurement
- TDC events
- DLD events

For a code example, see *User Callbacks Pipe*.

4.3.1 DLD applications

The callback function for receiving DLD events obtains an array of the struct sc_DldEvent, which is defined in the file scTDC_types.h as follows:

```
struct sc_DldEvent
{
  unsigned long long start_counter;
  unsigned long long time_tag;
  unsigned subdevice;
  unsigned channel;
  unsigned long long sum;
  unsigned short dif1;
```

(continues on next page)

(continued from previous page)

```
  unsigned short dif2;
  unsigned master_rst_counter;
  unsigned short adc;
  unsigned short signal1bit;
};
```

The meaning of the data fields for DLD events is given below:

start_counter : The start counter field of a detector event contains the number of start pulses applied as digital pulses on the "Start In" input of the TDC (which provides the time reference for time-resolved measurements). Time-resolved measurements require "Ext_Gpx_Start=YES" in the configuration/"ini" file of your TDC. Additionally, "ExtendedTimeRange = YES" is required to get access to (non-zero) start counter values.

time_tag : The time tag value is related to the "TAG In" input of the TDC. In the TDC configuration ("ini") file, the configuration parameter "TimeTag" can be set to integer numbers from 0 to 6. Not all types of TDCs support the full range of modes — modes 0,1,2 are supported by most models built for DLD applications, while modes 3-6 are reserved to advanced models. "TimeTag=0" disables the "TAG In" input. "TimeTag=1" means that the time tag value delivered together with a detector event counts the number of 12.5 ns cycles elapsed since the last digital pulse on the "TAG In" input (timer mode). "TimeTag=2" means that the time tag value delivered together with a detector event counts the number of digital pulses on the "TAG In" input since the start of the current exposure. "TimeTag=3" and "TimeTag=4" have identical meaning and enable the usage of the ADC input and the State input, but disable the "TAG In". "TimeTag=5" and "TimeTag=6" correspond to "TimeTag=1" and "TimeTag=2" but enable simultaneous use of the "State" input.

subdevice : Indicates the number of the subdevice, from which the DLD event originated, if you have a device that internally combines several TDC units.

channel : For standard 2D delay-line detectors, the channel field of an event is always zero (or it may deliver numbers from 0-3 which have no significance). In case of specialized types of detectors with multiple segments, the channel may indicate the segment.

sum : the time value of events with respect to a start/reference pulse applied to the Start In input.

dif1, *dif2* : the detector coordinate x, y

master_rst_counter : the master rst counter field of a detector event counts the number of digital pulses applied to the "Master Reset Counter" input of the TDC. The value is only reset manually by the user or during initialization of the TDC. Manual reset is possible via the API function "int sc_tdc_zero_master_reset_counter (const int dd);"

adc : the ADC field of a detector event is a digital value corresponding to the analog voltage difference applied between the "ADC+" and "ADC-" (differential) inputs. The input is designed for voltage differences from -10 Volts to +10 Volts. The range from 0 to +10 Volt is mapped linearly to digital values 0 to 32767, and the range from -10 Volt to 0 V is mapped linearly to digital values from 32768 to 65536.

signal1bit : can be zero or one, corresponding to a low or high voltage level on the "State" input of the TDC. The value for the "State" input is only available if in the configuration ("ini") file of the TDC, the parameter "TimeTag" is set to one of 3, 4, 5, 6.

4.3.2 TDC applications

If you are using the user callbacks pipe with a stand-alone TDC, you will be interested in TDC events, instead. The callback function for receiving TDC events obtains an array of the struct sc_TdcEvent, which is defined in the file scTDC_types.h as follows:

```
struct sc_TdcEvent
{
  unsigned subdevice;
  unsigned channel;
  unsigned long long start_counter;
  unsigned long long time_tag;
```

(continues on next page)

(continued from previous page)

```
  unsigned long long time_data;
  unsigned long long sign_counter;
};
```

The meaning of the data fields for TDC events is given below:

subdevice: Indicates the number of the subdevice, from which the TDC event originated, if you have a device that internally combines several TDC units.

channel: The TDC channel on which the pulse was registered.

start_counter: the number of the start period in which the pulse was registered. The number is reset to zero at the start of a measurement.

time_data: the time of the event in number of time bins since the last start pulse

time_tag, *sign_counter*: the values in these fields may provide additional information related to respective inputs.

5 Recon*Flex*™ cameras

Many aspects of the scTDC SDK work very similar for ReconFlex™ cameras compared to the TDC and DLD products by Surface Concept GmbH. Initialization of the device, the general concept of pipes for receiving data, the starting of measurements, and deinitialization of the device work the same way. However, cameras offer a number of configuration options and operation modes which are controlled by camera-specific API functions.

5.1 Configuring camera parameters

5.1.1 Setting exposure (per frame) and number of frames for measurements

For ReconFlex™ cameras, measurements are not characterized by a single duration value. They can comprise a configurable number of camera frames, which means that the camera sensor may perform multiple exposures and accordingly be read out multiple times during a single measurement. Within one measurement, all camera frames are taken with the same exposure. The total duration of a measurement is still somewhat complex to estimate. In the simplest case, it is roughly the product of *number of frames* times the *exposure per frame*. However, if the exposure per frame is chosen very small and the hypothetical gap-less frame sequence exceeds the maximum frame rate of the camera, the firmware will instead drive the sequence of frame exposures with time gaps between the frames to the effect that the maximum possible frame rate is reached.

As a consequence, when calling `sc_tdc_start_measure2()`, the argument for the exposure time is ignored. The exposure time and number of frames for the camera is instead controlled by calling `sc_tdc_cam_set_exposure()` before starting the measurement, thereby implicitly defining the total duration of a measurement.

5.1.2 Selecting *raw image* mode or *blob* mode

A central aspect concerning configuration is whether the camera is set up to deliver *raw* images (as recorded by the sensor) or blob data (where intensity peaks in the raw image are recognized by the camera firmware and turned into a list of coordinates that is sent to the PC).

When the application initializes the camera, the selected mode depends on entries in the *ini file* (`BlobDifMinTop` and `BlobDifMinBottom` in the `[CMOS]` section). To change the mode, make sure, the camera is initialized and not currently in a measurement.

- To select the *raw image* mode, create a variable of type `sc_BlobParameters`, fill its contents as shown in the code below and call `sc_tdc_set_blob_parameters()`:

```
// assumes that an 'int dev_desc' variable contains the
// device descriptor of an initialized camera
```

(continues on next page)

(continued from previous page)

```
sc_BlobParameters p;
p.unbinning = 1;
p.dif_min_top = 0;
p.dif_min_bottom = 0;
p.z_scale_factor = 1.0;
int ret = sc_tdc_set_blob_parameters(dev_desc, &p);
if (ret < 0) {
  // an error occurred
}
```

- To select *blob* mode, do the same with values for `p.dif_min_top` and `p.dif_min_bottom` that are greater than 0:

```
sc_BlobParameters p;
p.unbinning = 1;
p.dif_min_top = 2;
p.dif_min_bottom = 5;
p.z_scale_factor = 1.0;
int ret = sc_tdc_set_blob_parameters(dev_desc, &p);
if (ret < 0) {
  // an error occurred
}
```

In these examples, `p.dif_min_top` and `p.dif_min_bottom` correspond to the *ini file* entries `BlobDifMinTop` and `BlobDifMinBottom`, respectively. Their allowed values range from 1 to 63 for blob mode, whereas the special value 0 turns blob mode off. The lower these values, the more blobs are found. For more details, see *Blob recognition criteria*.

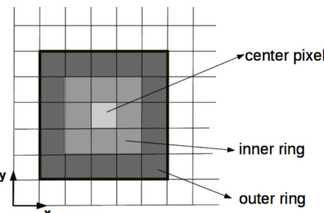
The blob recognition criteria also refer to `BlobRelax` (*ini file* parameter). After initialization of the camera, this parameter can be changed by application code, as follows (here, the parameter is set to 0):

```
// assume that dev_desc is a variable containing the device descriptor
sc_tdc_cam_set_parameter(dev_desc, "BlobRelax", "0");
```

The `sc_tdc_cam_set_parameter()` expects the parameter value in a string form (`char*`).

5.1.3 Blob recognition criteria

For blob recognition, the firmware of the camera scans through the raw image and evaluates a 5x5 square of intensities at every position. A set of conditions is checked to decide whether there is a blob at the center of the currently considered 5x5 square, where two rings of intensities are distinguished as shown in the following image:



- **Condition 1**: The grey scale value of the central pixel must be bigger than or equal to all 8 pixels of the inner ring.
- **Condition 2**: Each of the pixels in the inner ring must have grey scale values bigger than or equal to their neighbours that are part of the outer ring. This condition can be relaxed via the parameter `BlobRelax` as follows:
  - `BlobRelax = 0` : original behaviour, no relaxation

   - `BlobRelax = 1` : no comparison with the corners in the outer ring
   - `BlobRelax = 2` : only the 4 pixels in the outer ring lying on the horizontal and vertical axes are compared
   - `BlobRelax = 3` : condition 2 is not probed at all
- **Condition 3**: The grey scale value of the center pixel exceeds the average of the grey scale values of the inner ring by a difference at least as high as `BlobDifMinTop` (*ini file* parameter), or field `dif_min_top` in *sc_BlobParameters*.
- **Condition 4**: The average of the grey scale values of the inner ring exceeds the average of the outer ring by a difference at least as high as `BlobDifMinBottom` (*ini file* parameter), or field `dif_min_bottom` in *sc_BlobParameters*.

5.1.4 Setting a region of interest (ROI) on the hardware level

When recording images or blob data with a camera, not all of the sensor area may be of interest. Reducing the active area of the sensor enables access to higher frame rates and reduces data transfer (per recorded camera frame) from the camera to the PC. The reduced data transfer has the largest impact when the camera is operated in the *raw image* mode. The increased frame rates are enabled when reducing the number of lines (ROI height).

After initialization of the device, the ROI defaults to a setting defined by the *ini file* parameters `x_min`, `x_max`, `y_min`, `y_max` in the `[CMOS]` section.

To modify the region of interest, make sure the device is initialized and not currently in a measurement, then call *sc_tdc_cam_set_roi()*.

The region of interest may not always be set to exactly the boundaries specified in *sc_tdc_cam_set_roi()* – for example, if the request exceeded the available sensor area or due to technical limitations that prevent pixel-perfect selection. To query the actual active ROI after your set request, use *sc_tdc_cam_get_roi()*.

5.1.5 Selecting the dynamics range of intensities in raw images

The sensor of the camera can be configured to enable maximum intensities per pixel of either 255 (8-bit) or 4095 (12-bit). The *12-bit mode* allows for longer exposures without saturating pixels at the maximum intensity. The *8-bit mode* enables higher frame rates.

Note: Even in blob mode, the dynamics range has an impact, since the blob data is based on localizing intensity peaks in raw images (which happens in the camera). Even if you are not interested in the raw images, accurate blob positions require that the raw images are not overexposed.

The choice between *8-bit mode* and *12-bit mode* is fixed during initialization stage and cannot be modified afterwards. By default, the *ini file* parameters `BitMode` and `BitTransferMode` define which mode is active. To modify this choice from code without modification of the *ini file*, you can use the *override registry* interface:

```
int ovr = sc_tdc_overrides_create();
sc_tdc_overrides_add_entry(ovr, "CMOS", "BitMode", "8");
sc_tdc_overrides_add_entry(ovr, "CMOS", "BitTransferMode", "8");
sc_tdc_overrides_add_entry(ovr, "CMOS", "BitShift", "0");
// or: sc_tdc_overrides_add_entry(ovr, "CMOS", "BitMode", "12");
//     sc_tdc_overrides_add_entry(ovr, "CMOS", "BitTransferMode", "12");
//     sc_tdc_overrides_add_entry(ovr, "CMOS", "BitShift", "0");
int dd = sc_tdc_init_inifile_override("tdc_gpx3.ini", ovr);
sc_tdc_overrides_close(ovr);
// if dd >= 0, the camera is now initialized with the selected bit mode.
```

A mixed mode exists, where the sensor is configured for *12-bit mode* but the data transferred to the PC is cropped to *8-bit* intensity values. This can be done by dividing the sensor intensities by $2^4 = 16$, which maps the maximum possible intensities from *12-bit mode* to *8-bit mode* and reduces the resolution in graylevels. This choice corresponds to parameter values `BitMode = 12`, `BitTransferMode = 8`, `BitShift = 4`. This setting enables the same length of exposures as in 12-bit mode but reduces data transfer to the PC. Alternatively, the sensor intensity can be divided by a lower power of 2, if the expected intensity values are low enough. This is achieved by `BitShift` values less than 4.

5.2 Receiving Data

5.2.1 DLD_IMAGE_XY pipe versus PIPE_CAM_FRAMES pipe

The image pipe *DLD_IMAGE_XY*, originally aimed at delay-line detectors, can be used to deliver raw sensor images or images constructed from the list of blobs. This functionality is offered as a backwards compatible way for existing DLD applications that have been using this pipe before. Since DLD image pipes deliver only one data set per measurement, in general they add the intensities from multiple camera frames, or filter a subset of the camera frames. The configuration parameters of these image pipes do not contain a field for camera frames. Instead the time-related parameters are reinterpreted as parameters relating to the frame index. If the camera is configured to record 5 frames in a single measurement, we label the frames by indices 0, 1, 2, 3, 4. Setting the `roi.offset.time = 1` and the `roi.size.time = 2` would mean that the intensities in the frames with indices 1 and 2 are added up and the resulting image is delivered at the end of the measurement. The remaining camera frames 0, 3, 4 are discarded in that case.

The image pipe *DLD_IMAGE_XY* introduces a considerable computational overhead, when used for raw images, since this pipe needs to implement the behaviour suggested by its configuration parameters — such as imposing a region of interest in addition to the region of interest defined at the hardware level of the camera sensor. A **more performant interface** for receiving raw image data is the *PIPE_CAM_FRAMES* pipe. This pipe delivers image data of individual camera frames while they are being reconstructed from the hardware data stream sent to the PC. Additionally, this pipe delivers meta information for every camera frame. Opening and reading of this pipe has very little impact on computational effort.

5.2.2 User callbacks interface versus PIPE_CAM_BLOBS pipe

The *user callbacks interface*, originally aimed at delay-line detectors, can be used to deliver lists of blob data for example in existing applications which already integrated this interface. Here, the `sc_DldEvent` is reinterpreted as a blob event. The fields `dif1` and `dif2` contain the blob coordinate. The field `adc` contains the digitized value of the analog voltage on the *ADC* hardware input which is updated once per frame. The field `sum` contains the index of the camera frame. The field `time_tag` contains the time stamp of the camera frame. Other fields are without meaning.

The user callbacks interface while not being the most efficient way of transporting blob data, offers decent performance. Its limitation is the data type of the blob coordinates which is an unsigned integer type. ReconFlex™ cameras of the model variant 'S' offer precision of blob coordinates beyond the pixel grid of the camera sensor by evaluating a blob position from its intensity distribution in the raw image. The most natural way to express the blob coordinates is therefore by using floating-point number types. With the user callbacks interface, the loss of precision can be worked around by configuring an upscaling option (see `sc_BlobParameters` and `sc_tdc_set_blob_parameters()`). In that case, the floating-point blob coordinates are multiplied by the upscaling factor before being stored in the `sc_DldEvent` structure.

The replacement for the user callbacks interface is the *PIPE_CAM_BLOBS* pipe.

5.2.3 Combining PIPE_CAM_FRAMES and PIPE_CAM_BLOBS pipe

The above-mentioned pipe types are designed such that they can be read in a synchronized fashion. They both deliver one data set per camera frame at the moment that the hardware protocol stream, internally decoded by the scTDC library, has reached the end of a camera frame. Consequently, the calls to `sc_pipe_read2()` for these two pipes can be put in sequence into a loop, and as long as both pipes are being read the same number of times, their returned data belongs to the same camera frame. This kind of reading pattern is necessary to associate blob data with particular camera frames and measurements, as well as the meta data associated with each frame, such as ADC values and frame time stamps. The example code *Camera pipes for frame meta info, raw images and blobs* reads both pipe types in a synchronized way.

6 Code Examples

6.1 Time Histogram Pipe (stand-alone TDC)

The following example demonstrates usage of a 1D time histogram for the TDC channel input **0** of a stand-alone TDC:

```
// requires C99 standard or later
#include <scTDC.h>
#include <stdio.h>

int main()
{
  int dd = sc_tdc_init_inifile("tdc_gpx3.ini");
  if (dd < 0) { // unable to initialize hardware. dd contains error code
    char error_description[ERRSTRLEN];
    sc_get_err_msg(dd, error_description);
    puts(error_description);
    return dd;
  } else { // dd is device descriptor which is used in all other functions
    double tdc_binsize;
    int ret = sc_tdc_get_binsize2(dd, &tdc_binsize);
    if (ret < 0) { //if there is error happened.
      puts("could not get binsize");
      return ret;
    }
    printf("tdc binsize is %lf\n", tdc_binsize);

    // now open a tdc_histo pipe

    struct sc_pipe_tdc_histo_params_t params;
    params.depth = BS32; // 32 bit per time channel (point) in the histogram
    params.channel = 0; // pipe for channel #0 is requested
    params.modulo = 0; // modulo is off
    params.binning = 4; // histogram binning is set to 4
    params.offset = 1000; // histogram starts from the 1000 time bins (see sc_tdc_get_
↪binsize2()).
    params.size = 2000; // histogram size is 2000 time bins (but note binning)!
    params.accumulation_ms = 0; // accumulation is off
    params.allocator_owner = NULL; // parameter for allocator cbf
    params.allocator_cb = NULL; // internal allocator is used
```

(continues on next page)

(continued from previous page)

```
    int pipe_id = sc_pipe_open2(dd, TDC_HISTO, (void *)&params);
    if (pipe_id < 0) { //pipe_id contains error code
      char error_description[ERRSTRLEN];
      sc_get_err_msg(pipe_id, error_description);
      puts(error_description);
      return pipe_id;
    }

    sc_tdc_start_measure2(dd, 1000); // start a measurement for 1000 milliseconds

    unsigned *tdc_histo;
    ret = sc_pipe_read2(dd, pipe_id, (void *)&tdc_histo, -1); //after the call
    // tdc_histo pointer will point to the histogram data. The buffer for the
    // histogram data will be allocated by an internal allocator and will be
    // destroyed during the next call to sc_pipe_read2.
    // -1 in the last argument means to wait infinitely (2^32 milliseconds).

    if (ret < 0) { // note that this could be timeout as well
      char error_description[ERRSTRLEN];
      sc_get_err_msg(pipe_id, error_description);
      puts(error_description);
      return ret;
    }

    // Now, we have the data and may process it or visualize it.
    // For example, tdc_histo[0] is the first histogram element,
    // that contains the summed number of detected pulses for the
    // basic time bins '4000', '4001', '4002', '4003' --- since
    // the pipe was configured with binning 4 and offset 1000.


    sc_pipe_close2(dd, pipe_id);

    // Here, the data pipe is closed. NOTE: tdc_histo now points to an
    // invalid memory region, that must not be accessed anymore. Any access
    // to the tdc_data has to be performed before calling sc_pipe_close2().
    // If necessary, make a copy before calling sc_pipe_close2().

    sc_tdc_deinit2(dd); // Release hardware and resources.
    return 0;
  }
}
```

6.2 Image Pipe

The following example demonstrates usage of a data pipe for images (images are targeted towards delay-line detector and camera applications and do not receive data, if a *stand-alone* TDC is used):

```
int image2d_ex() // 2d image example.
{
  const uint32_t size_x = 512;
  const uint32_t size_y = 512;
```

(continues on next page)

(continued from previous page)

```
// initialize the hardware and get a device descriptor
int dd = sc_tdc_init_inifile("tdc_gpx3.ini");
if (dd < 0) {
  char error_description[ERRSTRLEN];
  sc_get_err_msg(dd, error_description);
  printf("error! code: %d, message: %s\n", dd, error_description);
  return dd;
}

struct sc_pipe_dld_image_xy_params_t prms;
memset(&prms, 0, sizeof(prms));
prms.depth = BS32; //4 bytes per pixel in the image
prms.channel = -1; //all channels together
prms.binning = {1, 1, 1};
prms.roi = {{0,0,0}, {size_x, size_y, -1}};

// configure an "image pipe" and store the pipe id in "pd"
int pd = sc_pipe_open2(dd, DLD_IMAGE_XY, (void *)&prms);
if (pd < 0) { // check for an error
  char error_description[ERRSTRLEN];
  sc_get_err_msg(pd, error_description);
  printf("error! code: %d, message: %s\n", pd, error_description);
  sc_tdc_deinit2(dd);
  return pd;
}

int ret = sc_tdc_start_measure2(dd, 200); //start 200 ms measurement
if (ret < 0) {
  char error_description[ERRSTRLEN];
  sc_get_err_msg(ret, error_description);
  printf("error! code: %d, message: %s\n", ret, error_description);
  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);
  return ret;
}

uint32_t *image;
// The following sc_pipe_read2(...) call blocks (waits) until the measurement
// is finished. The image variable is set to the image data buffer, allocated
// by the library ("internal memory allocation"). The size of the image buffer
// in bytes is roi_x  * roi_y * 4, in this example 512 * 512 * 4 (see prms.roi
// and prms.depth settings).
// Deallocation happens when the next call to sc_pipe_read2(), sc_pipe_close2()
// or sc_tdc_deinit2() is made.
ret = sc_pipe_read2(dd, pd, (void **) &image, UINT32_MAX);
if (ret < 0) {
  char error_description[ERRSTRLEN];
  sc_get_err_msg(ret, error_description);
  printf("error! code: %d, message: %s\n", ret, error_description);
  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);
  return ret;
}

// Here, the application can process the image data.
for (size_t i=0; i<size_x * size_y; ++i) {
```

(continues on next page)

(continued from previous page)

```
    fprintf(stderr, "%08x\n", image[i]);
  }

  // close the pipe (this invalidates the image pointer) and the device
  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);
  fprintf(stderr, "\n");
  return 0;
}
```

6.3 External Memory Allocation

Example using a statistics pipe with external allocator:

```
class Allocator
{
    // This is the allocator class which stores all statistics in the mem_chunks_.
    // Deallocation happens when the object is destroyed.
    std::list <std::unique_ptr <unsigned char []> > mem_chunks_;
    const size_t chunk_size_;
  public:
    Allocator (size_t s) :chunk_size_(s) {}
    static int pre_alloc(void *p, void **u) {
      return (static_cast <Allocator *> (p))->alloc(u);
    }

    int alloc(void **u) {
      std::unique_ptr <unsigned char []> chunk(new unsigned char [chunk_size_]);
      memset(&(chunk[0]), 0, chunk_size_); // initialize memory to all zeros
      *u = &(chunk[0]);
      mem_chunks_.push_back(std::move(chunk));
      return 0;
    }
};

int statistics_pipe_ex()
{
  // initialize the hardware and get device descriptor
  int dd = sc_tdc_init_inifile("tdc_gpx3.ini");
  if (dd < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(dd, error_description);
    printf("error! code: %d, message: %s\n", dd, error_description);
    return dd;
  }

  sc_pipe_statistics_params_t prms;
  memset(&prms, 0, sizeof(prms));
  Allocator mem(sizeof(statistics_t));
  prms.allocator_owner = static_cast <void *> (&mem);
  prms.allocator_cb = &(mem.pre_alloc);

  int pd = sc_pipe_open2(dd, STATISTICS, (void *)&prms);
  if (pd < 0) {
```

(continues on next page)

(continued from previous page)

```
    char error_description[ERRSTRLEN];
    sc_get_err_msg(pd, error_description);
    printf("error! code: %d, message: %s\n", pd, error_description);
    sc_tdc_deinit(dd);
    return pd;
  }

  int ret = sc_tdc_start_measure2(dd, 200); // start 200 ms measure
  if (ret < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(ret, error_description);
    printf("error! code: %d, message: %s\n", ret, error_description);
    sc_pipe_close2(dd, pd);
    sc_tdc_deinit(dd);
    return ret;
  }

  statistics_t *stat;

  // During measurement, the scTDC library calls the Allocator::pre_alloc
  // function, which allocates memory to hold one instance of a statistics_t
  // record, saves the memory block in the list and returns it to the library.
  // The next function blocks until the measurement is finished. The application
  // gets the pointer to the statistics data from the sc_pipe_read2() function.
  // The memory is deallocated when the 'mem' variable is destroyed (which
  // happens automatically at the end of main()). If the application calls
  // sc_tdc_start_measure2() and sc_pipe_read2() several times, here the 'mem'
  // object accumulates memory chunks in the list.
  // An alternative implementation of the Allocator class could also choose
  // to just allocate a single memory buffer and return this buffer from the
  // allocator callback multiple times. The library modifies the buffer by
  // adding values to its elements, rather than overwriting elements with new
  // values. This results in an accumulating effect across multiple measurements
  // (the library does not reset the buffer to zero, so the zeroing remains
  // under the control of the application developer).
  // This works for images and histograms as well.

  int ret = sc_pipe_read2(dd, pd, (void *)&stat, UINT32_MAX);
  if (ret < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(ret, error_description);
    printf("error! code: %d, message: %s\n", ret, error_description);
    sc_pipe_close2(dd, pd);
    sc_tdc_deinit(dd);
    return ret;
  }

  // Here we can do something with the statistics data
  printf("counts_read[0][0] = %d\n", stat->counts_read[0][0]);

  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);

  // Here, the statistics data is still accessible because the 'mem' object is
  // still on the stack.
  // In case of 'internal' memory allocation, the pointer to the statistics data
```

(continues on next page)

(continued from previous page)

```
  // would be invalid at this point.
  return 0;
}
```

6.4 User Callbacks Pipe

The following example shows how to set up the user callbacks pipe to process TDC or DLD data in a sequence-of-events form:

```
#include <stdio.h>
#include <stdlib.h>
#include <scTDC.h>

struct sc_DeviceProperties3 sizes;

/* Include an actual Semaphore implementation, such as in
 * https://github.com/preshing/cpp11-on-multicore/blob/master/common/sema.h
 */
class Semaphore;
Semaphore sem;

void cb_start(void *p) {
  /* this function gets called every time a measurement starts */
}

void cb_end(void *p) {
  /* this function gets called every time a measurement finishes */
  sem.signal();
}

void cb_millis(void *p) {
  /* this function gets called every time a millisecond has ellapsed as
   * tracked by the hardware */
}

void cb_stat(void *p, const struct statistics_t *stat) {
  /* this function gets called every time statistics data is received,
   * usually at the end of every measurement, but before the end-of-measurement
   * callback */
}

void cb_tdc_event
(void *priv,
const struct sc_TdcEvent *const event_array,
size_t event_array_len)
{
  const char *buffer = (const char *) event_array;
  size_t j;
  for (j=0; j<event_array_len; ++j) {
    const struct sc_TdcEvent *obj =
      (const struct sc_TdcEvent *)(buffer + j * sizes.tdc_event_size);
    /* insert code here, that uses the TDC event data.
     * obj->channel, obj->time_data ... contain information about
     * the j-th TDC event provided during this call.
```

(continues on next page)

(continued from previous page)

```
    */
  }
}

void cb_dld_event
(void *priv,
const struct sc_DldEvent *const event_array,
size_t event_array_len)
{
  const char *buffer = (const char *) event_array;
  size_t j;
  for (j=0; j<event_array_len; ++j) {
    const struct sc_DldEvent *obj =
      (const struct sc_DldEvent *)(buffer + j * sizes.tdc_event_size);
    /* insert code here, that uses the DLD event data.
     * obj->dif1, obj->dif2, obj->sum ... contain information about
     * the j-th DLD event provided during this call. */
  }
}


int main()
{
  int dd;
  int ret;
  struct PrivData priv_data;
  char *buffer;
  struct sc_pipe_callbacks *cbs;
  struct sc_pipe_callback_params_t params;
  int pd;

  dd = sc_tdc_init_inifile("tdc_gpx3.ini");
  if (dd < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(dd, error_description);
    printf("error! code: %d, message: %s\n", dd, error_description);
    return dd;
  }

  ret = sc_tdc_get_device_properties(dd, 3, &sizes);
  if (ret < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(ret, error_description);
    printf("error! code: %d, message: %s\n", ret, error_description);
    return ret;
  }

  buffer = calloc(1, sizes.user_callback_size);
  cbs = (struct sc_pipe_callbacks *)buffer;
  cbs->priv = &priv_data;
  cbs->start_of_measure = cb_start;
  cbs->end_of_measure = cb_end;
  cbs->millisecond_countup = cb_millis;
  cbs->statistics = cb_stat;
  cbs->tdc_event = cb_tdc_event;
  cbs->dld_event = cb_dld_event;
```

(continues on next page)

(continued from previous page)

```
  params.callbacks = cbs;

  pd = sc_pipe_open2(dd, USER_CALLBACKS, &params);
  if (pd < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(pd, error_description);
    printf("error! code: %d, message: %s\n", pd, error_description);
    return pd;
  }

  free(buffer);

  ret = sc_tdc_start_measure2(dd, 1000);
  if (ret < 0) {
    char error_description[ERRSTRLEN];
    sc_get_err_msg(ret, error_description);
    printf("error! code: %d, message: %s\n", ret, error_description);
    return dd;
  }

  /* Wait until the semaphore is signalled, which happens in our callback for
   * the end-of-measurement event */
  sem.wait();

  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);

  return 0;
}
```

6.5 Camera pipes for frame meta info, raw images and blobs

The following example shows the usage of the pipe types PIPE_CAM_FRAMES and PIPE_CAM_BLOBS for camera applications which enable reading of per-frame meta information, raw image data, and blob data:

```
#include <scTDC.h>
#include <scTDC_cam.h>
#include <scTDC_cam_types.h>
#include <scTDC_types.h>
#include <scTDC_error_codes.h>
#include <iostream>
#include <chrono>
#include "sema.h"
// Semaphore implementation available at
// https://github.com/preshing/cpp11-on-multicore/blob/master/common/sema.h

class TestCamFramesBlobs {
  Semaphore sema;
  bool end_of_meas = false;
public:
  enum { EXPOSURE_MICROSECS = 1000, NUMBER_OF_FRAMES = 500 };
  static void static_complete_cb(void* p, int reason) {
    static_cast<TestCamFramesBlobs*>(p)->complete_cb(reason);
  }
```

(continues on next page)

(continued from previous page)

```
void complete_cb(int reason) {
  if (reason != 4) {
    end_of_meas = true;
    sema.signal();
  }
}

void run() {
  auto dd = sc_tdc_init_inifile("tdc_gpx3.ini");
  if (dd < 0) {
    char buf[255];
    sc_get_err_msg(dd, buf);
    std::cout << "error during initialization : " << buf << "\n";
    return;
  }
  sc_tdc_cam_set_exposure(dd, EXPOSURE_MICROSECS, NUMBER_OF_FRAMES);
  auto pipedesc = sc_pipe_open2(dd, PIPE_CAM_FRAMES, nullptr);
  if (pipedesc < 0) {
    char buf[255];
    sc_get_err_msg(pipedesc, buf);
    std::cout << "error during pipe_open for frames: " << buf << "\n";
    sc_tdc_deinit2(dd);
    return;
  }

  auto pipedesc2 = sc_pipe_open2(dd, PIPE_CAM_BLOBS, nullptr);
  if (pipedesc2 < 0) {
    char buf[255];
    sc_get_err_msg(pipedesc2, buf);
    std::cout << "error during pipe_open for blobs: " << buf << "\n";
    sc_pipe_close2(dd, pipedesc);
    sc_tdc_deinit2(dd);
    return;
  }

  auto ret2 = sc_tdc_set_complete_callback2(dd, this, static_complete_cb);
  if (ret2 < 0) {
    char buf[255];
    sc_get_err_msg(ret2, buf);
    std::cout << "error while setting complete cb : " << buf << "\n";
  }

  auto ret3 = sc_tdc_start_measure2(dd, 100);
  if (ret3 < 0) {
    char buf[255];
    sc_get_err_msg(ret3, buf);
    std::cout << "error during start of measurement : " << buf << "\n";
  }

  auto t1 = std::chrono::steady_clock::now();

  while(true) {
    void* buf_frames = nullptr;
    void* buf_blobs = nullptr;
    int ret = sc_pipe_read2(dd, pipedesc, &buf_frames, 50);
    if (ret == SC_TDC_ERR_TIMEOUT) {
```

(continues on next page)

(continued from previous page)

```
        if (end_of_meas) {
          break;
        }
        else {
          continue;
        }
      }
      else if (ret < 0) {
        break;
      }
      int ret2 = sc_pipe_read2(dd, pipedesc2, &buf_blobs, 1);
      if (buf_frames != nullptr) {
        const auto& meta = *static_cast<sc_cam_frame_meta_t*>(buf_frames);
        std::cout << "frame #" << meta.frame_idx << " width=" << meta.width
                  << " height=" << meta.height << " adc=" << meta.adc
                  << " time=" << meta.frame_time << " bpp="
                  << ((meta.pixelformat == SC_CAM_PIXELFORMAT_UINT8) ? 8 : 16)
                  << " last=" << ((meta.flags & SC_CAM_FRAME_IS_LAST_FRAME) > 0)
                  << "\n";

      }
      else {
        std::cout << "FRAME MISSING!" << std::endl; // this should not happen
      }
      if (buf_blobs != nullptr) {
        const auto& metablobs = *static_cast<sc_cam_blob_meta_t*>(buf_blobs);
        std::cout << "number of blobs: " << metablobs.nr_blobs << "\n";
        if (metablobs.nr_blobs > 0) {
          const auto* blobs =
            reinterpret_cast<sc_cam_blob_position_t*>(
              static_cast<char*>(buf_blobs) + metablobs.data_offset);
          std::cout << " blob 0 : x = " << blobs[0].x << " y = " << blobs[0].y
                    << "\n";
        }
      }
      else {
        std::cout << "no blobs\n";
      }
      if (ret >= 0 && ret2 < 0) {
        std::cout << "FRAME BUT NO BLOB!\n";
      }
    }

    sema.wait();
    std::cout << "duration: "
      << std::chrono::duration_cast<std::chrono::milliseconds>(
           std::chrono::steady_clock::now() - t1).count() << " ms\n";

    sc_pipe_close2(dd, pipedesc);
    sc_tdc_deinit2(dd);
  }
};


int main() {
  TestCamFramesBlobs t;
```

(continues on next page)

(continued from previous page)

```
  t.run();
}
```

7 Old API notes

An older version of the scTDC library was providing a set of functions which did not include the possibility to initialize and use more than one device by the same application, simultaneously. For many of these older functions, an equivalent function with a suffix “2” in the name has been added. These newer functions contain a device descriptor argument. The older set of functions has been moved into a different header file (“scTDC_deprecated.h”). They should not be used for developing new applications (even if you don’t plan to use more than one device).

8 scTDC library API

8.1 Page Hierarchy

8.2 Class Hierarchy

8.3 File Hierarchy

8.4 Full API

8.4.1 Classes and Structs

Struct roi_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **roi_t**

Region of interest in a three-dimensional coordinate system spanned by detector position (x, y) and the time coordinate. Offsets mark the margins at the lower coordinate ends, sizes define the extension towards the higher coordinate end.

Public Members

struct *sc3d_t* **offset**

Roi offset

struct *sc3du_t* **size**

Roi size

Struct sc3d_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc3d_t**

Signed 3d point.

Public Members

int **x**

x coordinate.

int **y**

y coordinate.

long long **time**

time coordinate.

Struct sc3du_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc3du_t**

Unsigned 3d point.

Public Members

unsigned int **x**

x coordinate.

unsigned int **y**

y coordinate.

unsigned long long **time**

time coordinate.

Struct sc_BlobParameters

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_BlobParameters**

Blob parameters structure.

Public Members

unsigned int **unbinning**

scales blob coordinates by a power of 2

unsigned int **dif_min_top**

if >= 0, activate blob mode, threshold condition for blob recognition

unsigned int **dif_min_bottom**

if >= 0, activate blob mode, threshold condition for blob recognition

double **z_scale_factor**

for future use, set it to 1.0, for now

Struct sc_cam_blob_meta_t

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_cam_blob_meta_t**

Public Members

unsigned **data_offset**

memory address offset to the blob data

unsigned **nr_blobs**

number of blobs

Struct sc_cam_blob_position_t

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_cam_blob_position_t**

Public Members

float **x**

float **y**

Struct sc_cam_frame_meta_t

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_cam_frame_meta_t**

Public Members

unsigned **data_offset**

memory address offset to the image data

unsigned **frame_idx**

index of frame within measurement

unsigned long long **frame_time**

time stamp of the frame

unsigned short **width**

width of the image / currently set ROI

unsigned short **height**

height of the image / currently set ROI

unsigned short **roi_offset_x**

horizontal position of the ROI on sensor

unsigned short **roi_offset_y**

vertical position of the ROI on sensor

unsigned short **adc**

ADC value, digitized voltage on ADC hardware input

unsigned char **pixelformat**

see *sc_cam_pixelformat_t*

unsigned char **flags**

see *sc_cam_frame_meta_flags_t*

unsigned char **reserved**[4]

no data, inserted for memory alignment

Struct sc_CamProperties1

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_CamProperties1**

Public Members

int **supports_blob**

if > 0, the camera supports blob recognition

int **supports_upscaling**

if > 0, the camera supports blob coordinates with resolution beyond the pixel grid of the sensor

int **supports_adc**

if > 0, the camera supports an ADC input

Struct sc_CamProperties2

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_CamProperties2**

Public Members

int **supports_convolution_mask**

Struct sc_CamProperties3

- Defined in file_scTDC_cam_types.h

Struct Documentation

struct **sc_CamProperties3**

Public Members

int **sensor_max_intensity**

the maximum intensity level on the sensor according to the sensor type and configuration (camera frames read by the application may deliver down-scaled intensities compared to what is captured by the sensor due to the BitShift parameter in the [CMOS] section of the ini file

int **frame_max_intensity**

the maximum intensity level delivered in raw image camera frames

Struct sc_CmosSmootherParameters

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_CmosSmootherParameters**

Cmos and Smoother parameters structure.

Public Types

enum **sc_ShutterMode**

Values:

enumerator **FULLY_EXTERNAL**

start and stop of frames controlled by hardware input ("wire")

enumerator **IMMEDIATE_START_INTERNAL_TIMER**

fully software-controlled frames

enumerator **IMMEDIATE_START_INTERNAL_TIMER_MULTIPLE_SLOPES**

not in use anymore

enumerator **EXTERNAL_START_INTERNAL_TIMER**
start of frames by wire, stop by software

enumerator **EXTERNAL_START_INTERNAL_TIMER_MULTIPLE_SLOPES**
not in use anymore

enumerator **IMMEDIATE_START_EXTERNAL_FINISH**
start of frames by software, stop by wire

Public Members

enum *sc_CmosSmootherParameters*::*sc_ShutterMode* **shutter_mode**
controls triggering of camera frames by hardware input

unsigned int **single_slope_us**
the exposure of a single camera frame

unsigned int **dual_slope_us**
not in use anymore

unsigned int **triple_slope_us**
not in use anymore

unsigned int **frame_count**
sets the number of frames per measurement

unsigned int **analog_gain**
the analog gain parameter of the sensor

double **digital_gain**
not in use anymore

unsigned int **black_offset**
the black offset parameter of the sensor

int **black_cal_offset**
not in use anymore

unsigned int **smoother_shift1**
intensity scale-down after application of smoother_pixel_mask1

unsigned char **smoother_pixel_mask1**[8][8]
first smoother mask

unsigned int **smoother_shift2**
intensity scale-down after application of smoother_pixel_mask2

unsigned char **smoother_pixel_mask2**[8][8]

second smoother mask

unsigned char **white_pixel_min**

threshold for white pixel suppression

Struct sc_ConfigLine

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_ConfigLine**

Public Members

const char ***section**

const char ***key**

const char ***value**

Struct sc_DeviceProperties1

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DeviceProperties1**

Device Properties 1.

If mentioned below parameters are not available they are being set to 0. The x and y fields in detector_size report the size of the aperture, scaled by StretchX/Y + HardwareBinningX/Y parameters from the ini file — to the effect, that opening a DLD_IMAGE_XY histogram with the sizes for x and y set to detector_size.x and detector_size.y display the full image as cropped by the aperture settings. The pixel_size_x / pixel_size_y values are taken verbatim from the ini file and it is the responsibility of whoever adapts StretchX/Y and HardwareBinningX/Y to update these values, accordingly, such that they yield the physical size corresponding to one pixel in a DLD_IMAGE_XY histogram.

Public Members

struct *sc3du_t* **detector_size**

Physical detector size in pixels

double **pixel_size_x**

Pixel size in x direction in mm

double **pixel_size_y**

Pixel size in y direction in mm

double **pixel_size_t**

Pixel size in time direction in ns

Struct sc_DeviceProperties2

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DeviceProperties2**

Device Properties 2.

Public Members

int **tdc_channel_number**

Number of tdc channel device has

Struct sc_DeviceProperties3

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DeviceProperties3**

Device Properties 3.

Public Members

size_t **dld_event_size**

size of *sc_DldEvent* structure in bytes

size_t **tdc_event_size**

size of *sc_TdcEvent* structure in bytes

size_t **user_callback_size**

size of *sc_pipe_callbacks* structure in bytes

Struct sc_DeviceProperties4

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DeviceProperties4**

Device Properties 4.

Public Members

unsigned **auto_start_period**

automatically measured start period during initialization in units of time bins of the TDC / DLD

unsigned **auto_modulo**

automatically measured modulo during initialization

Struct sc_DeviceProperties5

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DeviceProperties5**

Device Properties 5.

Public Members

unsigned **tag_period**

averaged tag period (over ~1.6 seconds) when tag is used as a timer, given in “tag units” (12.5 ns)

Struct sc_dld_device_statistics_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_dld_device_statistics_t**

Public Members

unsigned long long **events_found**

unsigned long long **events_in_roi**

unsigned long long **events_received**

unsigned long long **reserved**[5]

Struct sc_DldEvent

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_DldEvent**

Dld data received from device.

Public Members

unsigned long long **start_counter**

Start pulse counter.

unsigned long long **time_tag**

The 'tag' value related to the 'Tag In' hardware input .

unsigned **subdevice**

Subdevice where DLD event occurred.

unsigned **channel**

Often unused, enumerates segments in multi-segment detectors

unsigned long long **sum**

Time of the DLD event in multiple of time bins referred to the last start pulse.

unsigned short **dif1**

X coordinate of dld event.

unsigned short **dif2**

Y coordinate of dld event.

unsigned **master_rst_counter**

counts up when a pulse is applied to the respective hardware input

unsigned short **adc**

the digitized 16-bit value of the ADC input

unsigned short **signal1bit**

either 0 or 1 depending on the voltage level applied to the State input

Struct sc_flimTriggersCounters

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_flimTriggersCounters**

Public Members

unsigned long long **pixelTriggerCounter**

unsigned long long **lineTriggerCounter**

unsigned long long **frameTriggerCounter**

Struct sc_Logger

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_Logger**

Logger descriptor used for debug.

The structure

See also:

sc_dbg_set_logger()

Public Members

void ***private_data**
Private data of the external logger.

void (***do_log**)(void *pd, const char *sender, const char *msg)
Logger callback function.

Param pd private_data field.

Param sender Sender of the debug message to be logger.

Param msg Message itself.

Struct sc_pipe_buf_callback_args

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_buf_callback_args**
Structure that is passed into a callback function for a BUFFERED_DATA_CALLBACKS pipe.

Any of the pointer members may be null, indicating that no data is available.

See also:

struct *sc_pipe_buf_callbacks_params_t*.

Public Members

unsigned long long **event_index**
Index of the first event.

unsigned long long ***som_indices**
Start of measurement indices.

unsigned long long ***ms_indices**
Millisecond indices.

unsigned ***subdevice**
Subdevice values.

unsigned ***channel**
Channel values.

unsigned long long ***start_counter**
Start counter values.

unsigned *time_tag
: Time Tag values.

unsigned *dif1
: Dif1 values / x detector coord.

unsigned *dif2
: Dif2 values / y detector coord.

unsigned long long *time
: Time values.

unsigned *master_rst_counter
: Master reset counter values.

int *adc
: ADC values.

unsigned short *signal1bit
: State input values.

unsigned som_indices_len
: length of som_indices array.

unsigned ms_indices_len
: length of ms_indices array.

unsigned data_len
: length of each data array.

unsigned char reserved[12]
: future use.

Struct sc_pipe_buf_callbacks_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct sc_pipe_buf_callbacks_params_t

Parameter structure to be passed as the third argument in sc_pipe_open2 when creating a BUFFERED_DATA_CALLBACKS pipe.

This kind of pipe may be useful for language bindings where invocation of callback functions is slow (e.g. Python). The pipe issues callbacks only once a configurable minimum number of events has been buffered — so as to reduce the frequency of callbacks. Data is buffered in separate arrays of basic datatypes and the event data fields to be buffered can be selected.

See also:

enum sc_data_field_t, *BUFFERED_DATA_CALLBACKS*

Public Members

void *`priv`

Private data.

void (*`data`)(void *priv, const *sc_pipe_buf_callback_args**const)

Callback providing buffered data.

bool (*`end_of_measurement`)(void *priv)

Callback signalizing end of measurement. If callback returns true, all currently buffered data will be immediately transferred via an invocation of the 'data' callback.

unsigned `data_field_selection`

select which of the event data fields to buffer.

unsigned `max_buffered_data_len`

maximum number of entries per data field to be buffered.

int `dld_events`

if 0, buffer TDC events; if 1, buffer DLD events

int `version`

version must be set to zero (enables future extensions)

unsigned char `reserved`[24]

future use.

Struct sc_pipe_callback_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct `sc_pipe_callback_params_t`

Parameters for USER_CALLBACKS pipe type, which is to be passed by pointer into the *sc_pipe_open2()*.

Public Members

struct *sc_pipe_callbacks* ***callbacks**

Struct sc_pipe_callbacks

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_callbacks**

Set of callback functions provided by the application developer to be called from the scTDC library for a USER_CALLBACKS pipe. The callback functions handle various events and the reception of TDC or DLD data. They are called synchronously from the library thread that decodes the protocol stream from the hardware, thus preserving correct chronological order of data and events.

Any callback pointers may be set to zero if the application does not need information about one or another event or data. For example, the tdc_event callback can be set to zero for DLD applications. (version 1.3017.5 fixes crashes for the case where 'start_of_measure' is zero).

Callback functions must limit the amount of time they need to execute, since the internal decoding of the hardware protocol stream pauses until the callback function returns. For best performance, you may decide to put the data you are interested in into a memory buffer that can be processed by a different thread, afterwards.

Furthermore, it is generally not possible to start new measurements from within the 'end_of_measure' callback. To work around this limit, it is recommended to use some notification mechanism into your main thread to schedule the start of the next measurement.

Make sure, not to pass this struct directly into the *sc_pipe_open2()* function. The correct usage is to take the address of a *sc_pipe_callbacks* variable, set this address in the 'callbacks' field of a *sc_pipe_callback_params_t* variable and pass the address of this latter variable to *sc_pipe_open2()*.

See also:

struct *sc_DeviceProperties3*, struct *sc_pipe_callback_params_t*

Public Members

void ***priv**

Private user pointer that is passed back into the callback functions

void (***start_of_measure**)(void *priv)

Called when the start of a measurement appears in the hardware protocol stream.

void (***end_of_measure**)(void *priv)

Called when the end of a measurement appears in the hardware protocol stream.

void (***millisecond_countup**)(void *priv)

Called at points where the hardware recorded an ellapsed millisecond.

void (***statistics**)(void *priv, const struct *statistics_t* *stat)

Called when statistics info appears in the hardware protocol stream, usually at the end of measurements.

void (*`tdc_event`)(void *priv, const struct *sc_TdcEvent* *const event_array, size_t event_array_len)

Called for transmission of TDC events (event_array_len = number of events)

void (*`dld_event`)(void *priv, const struct *sc_DldEvent* *const event_array, size_t event_array_len)

Called for transmission of DLD events (event_array_len = number of events)

Struct sc_pipe_dld_image_3d_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct `sc_pipe_dld_image_3d_params_t`

Parameters for DLD_IMAGE_3D pipe type.

The size of one histogram in memory is roi.x * roi.y * roi.t * element_size, where element_size depends on the choice of depth. Please note, that modulo, binning and roi settings are applied in the same order like set in the structure. e.g. first modulo is applied, then binning and then roi.

Public Members

enum *bitsize_t* `depth`

Data type of histogram elements.

int `channel`

Filter by channel if >= 0 (-1 for normal detector)

unsigned long long `modulo`

if > 0, apply modulo operation to time value before inserting it into the histogram.

struct *sc3du_t* `binning`

x,y,time are divided by the respective binnings before adding into the histogram.

struct *roi_t* `roi`

Region of interest from which image will be built.

unsigned int `accumulation_ms`

Accumulation time.

void *`allocator_owner`

A pointer chosen by the user that gets passed back into the allocator_cb

int (*`allocator_cb`)(void*, void**)

User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_dld_image_xt_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_image_xt_params_t**

Parameters for DLD_IMAGE_XT pipe type.

The size of one histogram in memory is roi.x * roi.t * element_size bytes, where element_size depends on the choice of depth.

Public Members

enum *bitsize_t* **depth**

Data type of histogram elements.

int **channel**

Filter by channel if >= 0 (-1 for normal detector)

unsigned long long **modulo**

if > 0, apply modulo operation to time value before inserting it into the histogram.

struct *sc3du_t* **binning**

x,y,time are divided by the respective binnings before adding into the histogram.

struct *roi_t* **roi**

Region of interest from which image will be built.

unsigned int **accumulation_ms**

Accumulation time.

void ***allocator_owner**

A pointer chosen by the user that gets passed back into the allocator_cb

int (***allocator_cb**)(void*, void**)

User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_dld_image_xy_ext_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_image_xy_ext_params_t**

Public Members

struct *sc_pipe_dld_image_xy_params_t* **base**

void ***extension**

Struct sc_pipe_dld_image_xy_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_image_xy_params_t**

Parameters for DLD_IMAGE_XY pipe type.

The size of one histogram in memory is roi.x * roi.y * element_size bytes, where element_size depends on the choice of depth.

Public Members

enum *bitsize_t* **depth**

Data type of histogram elements.

int **channel**

Filter by channel if >= 0 (-1 for normal detector)

unsigned long long **modulo**

if > 0, apply modulo operation to time value before inserting it into the histogram.

struct *sc3du_t* **binning**

x,y,time are divided by the respective binnings before adding into the histogram.

struct *roi_t* **roi**

Region of interest from which image will be built.

unsigned int **accumulation_ms**

Accumulation time.

void ***allocator_owner**

A pointer chosen by the user that gets passed back into the allocator_cb

int (***allocator_cb**)(void*, void**)

User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_dld_image_yt_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_image_yt_params_t**

Parameters for DLD_IMAGE_YT pipe type.

The size of one histogram in memory is roi.y * roi.t * element_size bytes, where element_size depends on the choice of depth.

Public Members

enum *bitsize_t* **depth**

Data type of histogram elements.

int **channel**

Filter by channel if >= 0 (-1 for normal detector)

unsigned long long **modulo**

if > 0, apply modulo operation to time value before inserting it into the histogram.

struct *sc3du_t* **binning**

x,y,time are divided by the respective binnings before adding into the histogram.

struct *roi_t* **roi**

Region of interest from which image will be built.

unsigned int **accumulation_ms**

Accumulation time.

void ***allocator_owner**

A pointer chosen by the user that gets passed back into the allocator_cb

int (***allocator_cb**)(void*, void**)

User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_dld_stat_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_stat_params_t**

Public Members

void ***allocator_owner**

int (***allocator_cb**)(void*, void**)

int **device_number**

Struct sc_pipe_dld_sum_histo_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_dld_sum_histo_params_t**

Parameters for DLD_SUM_HISTO pipe type — a 1D histogram with a time axis integrating over a region of interest with respect to detector positions. The size of one histogram in memory is roi.t * depth bytes.

Public Members

enum *bitsize_t* **depth**

Data type of histogram elements.

int **channel**

Filter by channel if >= 0 (-1 for normal detector).

unsigned long long **modulo**

if > 0, apply modulo operation to time value before inserting it into the histogram.

struct *sc3du_t* **binning**

x,y,time are divided by the respective binnings before adding into the histogram.

struct *roi_t* **roi**

Region of interest from which histogram will be built.

unsigned int **accumulation_ms**

Accumulation time.

void ***allocator_owner**

A pointer chosen by the user that gets passed back into the allocator_cb

int (***allocator_cb**)(void*, void**)

User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_image_source

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_image_source**

Public Types

enum **sc_ImageSource**

Values:

enumerator **EVENTS**

enumerator **RAMDATA**

enumerator **BOTH**

Public Members

enum *sc_pipe_param_extension_type* **type**

void ***extension**

enum *sc_pipe_image_source*::*sc_ImageSource* **source**

Struct sc_pipe_statistics_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_statistics_params_t**

Parameters for STATISTICS pipe type.

This pipes delivers data of the type struct *statistics_t*. The memory requirement for each such struct is sizeof(*statistics_t*) = 1024 bytes.

Public Members

void ***allocator_owner**

int (***allocator_cb**)(void*, void**)

Used to allocate memory for data. If NULL - direct memory allocation. allocator_owner field will used as first argument during the call.

Struct sc_pipe_tdc_histo_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_tdc_histo_params_t**

Parameters for TDC_HISTO pipe type.

The size of one histogram in memory is size * depth bytes.

Public Members

enum *bitsize_t* **depth**

Bits per histogram element in memory.

unsigned int **channel**

Channel is used to build histogram.

unsigned long long **modulo**

if > 0, apply modulo operation to time value before inserting it into the histogram.

unsigned int **binning**

Histogram time binning.

unsigned long long **offset**

Histogram start offset in time bins (see *sc_tdc_get_binsize2()*).

unsigned int **size**

Histogram size in time bins (see above).

unsigned int **accumulation_ms**

Accumulation time.

void ***allocator_owner**

int (***allocator_cb**)(void*, void**)

Parameter for the allocator_cb function. User-provided allocator function, that the library calls to allocate memory for data. If set to NULL, the library uses internal memory allocation. The allocator_owner pointer is passed into the allocator_cb as the first argument during the call.

Struct sc_pipe_tdc_stat_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_tdc_stat_params_t**

Public Members

void ***allocator_owner**

int (***allocator_cb**)(void*, void**)

int **channel_number**

Struct sc_pipe_used_mem_callbacks_params_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_pipe_used_mem_callbacks_params_t**

Public Members

void ***priv**

private data

void (***used_mem**)(void *priv, const unsigned used_mem_kb_value)

Struct sc_PipeCbf

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_PipeCbf**

Public Members

void (***cb**)(void*)

void ***private_data**

Struct sc_tdc_channel_statistics_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_tdc_channel_statistics_t**

Public Members

unsigned long long **counts_read**

unsigned long long **counts_received**

unsigned long long **counter**

unsigned long long **reserved**[5]

Struct sc_tdc_format

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_tdc_format**

Contains sizes and offsets of data bitfields.

Zero value of the field means that field is not present in the event.

Public Members

unsigned char **total_bits_length**

Length of one event in bits. Currently can be only 8, 16, 32 and 64

unsigned char **channel_offset**

Offset of channel field. Mostly used in tdc mode

unsigned char **channel_length**

Length of channel field. Mostly used in tdc mode. Channel field contains information in which channel of TDC event occured.

unsigned char **time_data_offset**

Offset of time data data field. Mostly used in tdc mode

unsigned char **time_data_length**

Length of time_data field. Mostly used in tdc mode. time_data field contains information about time when event occurs [binsize]

unsigned char **time_tag_offset**

unsigned char **time_tag_length**

unsigned char **start_counter_offset**

Offset of start_counter data field

unsigned char **start_counter_length**

Length of start_counter data field. start_counter data field contains information about start counter value. See documentation to the device for more info about start counter value

unsigned char **dif1_offset**

Offset of x coordinate of the event

unsigned char **dif1_length**

Length of x coordinate of the event. Mostly used in dld mode.

unsigned char **dif2_offset**

Offset of y coordinate of the event

unsigned char **dif2_length**

Length of y coordinate of the event. Mostly used in dld mode.

unsigned char **sum_offset**

Offset of time coordinate data field of the event in dld mode

unsigned char **sum_length**

Length of time coordinate data field of the event in dld mode.

unsigned char **sign_counter_offset**

unsigned char **sign_counter_length**

unsigned char **reserved**[14]

Reserved fields. Must not be used.

unsigned char **flow_control_flags**

Flow control flag data field.

Struct sc_TdcEvent

- Defined in file_scTDC_types.h

Struct Documentation

struct **sc_TdcEvent**

Tdc data received from device.

Public Members

unsigned **subdevice**

Subdevice where TDC event occurred.

unsigned **channel**

Tdc channel where TDC event occurred.

unsigned long long **start_counter**

Start pulse counter.

unsigned long long **time_tag**

The 'tag' value related to the 'Tag In' hardware input.

unsigned long long **time_data**

Time of the TDC event in multiple of time bins referred to the last start pulse.

unsigned long long **sign_counter**

counts up when a pulse is applied to the respective hardware input

Struct statistics_t

- Defined in file_scTDC_types.h

Struct Documentation

struct **statistics_t**

Measurement statistics. The first array index corresponds to the subdevice number, the second array index corresponds to channels. If there are no subdevices, data is found only for the first array index being set to 0.

Public Members

unsigned int **counts_read**[4][16]

Tdc data per channel recognized by fpga.

unsigned int **counts_received**[4][16]

Tdc data per channel transferred from fpga to cpu.

unsigned int **events_found**[4]

Dld events recognized in fpga.

unsigned int **events_in_roi**[4]

Dld events recognized in fpga which fits in hardware roi.

unsigned int **events_received**[4]

Dld events transferred from fpga to cpu.

unsigned int **counters**[4][16]

unsigned int **reserved**[52]

8.4.2 Enums

Enum bitsize_t

- Defined in file_scTDC_types.h

Enum Documentation

enum **bitsize_t**

Size and data type of pixel values (in images), or of histogram elements (in time histograms).

Values:

enumerator **BS8**

Unsigned 8-bit (1 byte) integer.

enumerator **BS16**

Unsigned 16-bit (2 bytes) integer.

enumerator **BS32**

Unsigned 32-bit (4 bytes) integer.

enumerator **BS64**

Unsigned 64-bit (8 bytes) integer.

enumerator **F32**

Single-precision IEEE 754 floating-point (4 bytes).

enumerator **F64**

Double-precision IEEE 754 floating-point (8 bytes).

Enum sc_cam_frame_meta_flags_t

- Defined in file_scTDC_cam_types.h

Enum Documentation

enum **sc_cam_frame_meta_flags_t**

Values:

enumerator **SC_CAM_FRAME_HAS_IMAGE_DATA**

if set, the #sc_cam_frame_meta_t block is followed by pixel data

enumerator **SC_CAM_FRAME_IS_LAST_FRAME**

if set, this frame is the last frame of the measurement

Enum sc_cam_pixelformat_t

- Defined in file_scTDC_cam_types.h

Enum Documentation

enum **sc_cam_pixelformat_t**

Values:

enumerator **SC_CAM_PIXELFORMAT_UINT8**

pixel values are unsigned 8-bit integers

enumerator **SC_CAM_PIXELFORMAT_UINT16**

pixel values are unsigned 16-bit integers

Enum sc_data_field_t

- Defined in file_scTDC_types.h

Enum Documentation

enum **sc_data_field_t**

Values:

enumerator **SC_DATA_FIELD_SUBDEVICE**

enumerator **SC_DATA_FIELD_CHANNEL**

enumerator **SC_DATA_FIELD_START_COUNTER**

enumerator **SC_DATA_FIELD_TIME_TAG**

enumerator **SC_DATA_FIELD_DIF1**

enumerator **SC_DATA_FIELD_DIF2**

enumerator **SC_DATA_FIELD_TIME**

enumerator **SC_DATA_FIELD_MASTER_RST_COUNTER**

enumerator **SC_DATA_FIELD_ADC**

enumerator **SC_DATA_FIELD_SIGNAL1BIT**

Enum sc_event_type_index

- Defined in file_scTDC_types.h

Enum Documentation

enum **sc_event_type_index**

Used as argument in functions *sc_tdc_is_event()*, *sc_tdc_is_event2()*.

Values:

enumerator **SC_TDC_SIGN_START**

Tdc event is start sign.

enumerator **SC_TDC_SIGN_MILLISEC**

Tdc event is millisecond sign.

enumerator **SC_TDC_SIGN_STAT**

Tdc event is beginning of statistics sign.

Enum sc_LoggerFacility

- Defined in file_scTDC_types.h

Enum Documentation

enum **sc_LoggerFacility**

Logging level.

Deprecated:

Is not used anymore.

See also:

sc_dbg_set_logger()

Values:

enumerator **UNUSED**

Enum sc_pipe_param_extension_type

- Defined in file_scTDC_types.h

Enum Documentation

enum **sc_pipe_param_extension_type**

Values:

enumerator **SC_PIPE_PARAM_EXTENSION_TYPE_IMAGE_SOURCE**

Enum sc_pipe_type_t

- Defined in file_scTDC_types.h

Enum Documentation

enum **sc_pipe_type_t**

Pipe type.

Values:

enumerator **TDC_HISTO**

Used to get pipe with tdc histo data

enumerator **DLD_IMAGE_XY**

Used to get 2d image data

enumerator **DLD_IMAGE_XT**

Used to get x-t image data

enumerator **DLD_IMAGE_YT**

Used to get y-t image data

enumerator **DLD_IMAGE_3D**

Used to get 3d (x,y,t cube) image data

enumerator **DLD_SUM_HISTO**

Used to get dld time histogram data

enumerator **STATISTICS**

Used to get statistics for last exposure

enumerator **TMSTAMP_TDC_HISTO**

enumerator **TDC_STATISTICS**

enumerator **DLD_STATISTICS**

enumerator **USER_CALLBACKS**

Used to receive lists of events (TDCs, DLDs, and cameras)

enumerator **DLD_IMAGE_XY_EXT**

Used to get 2d image data with extended parameter set

enumerator **BUFFERED_DATA_CALLBACKS**

Used by Python SDK, to receive lists of

events, reduces the frequency of callbacks

enumerator **PIPE_CAM_FRAMES**

Used to receive camera frame meta data, and raw image frame data, use nullptr as 3rd arg in *sc_pipe_open2()*

enumerator **PIPE_CAM_BLOBS**

Used to receive camera blob data, use nullptr as 3rd arg in *sc_pipe_open2()*

enumerator **USED_MEM_CALLBACKS**

Used to monitor hardware memory usage level

8.4.3 Functions

Function sc_dld_set_hardware_binning

- Defined in file_scTDC.h

Function Documentation

int **sc_dld_set_hardware_binning**(const int dd, const struct *sc3du_t* *binning)

Function sc_flim_get_counters

- Defined in file_scTDC.h

Function Documentation

int **sc_flim_get_counters**(const int dd, *sc_flimTriggersCounters**)

Function sc_get_err_msg

- Defined in file_scTDC.h

Function Documentation

void **sc_get_err_msg**(int err_code, char *err_msg)

Gives an error description in a text form.

Note: The err_msg array must not be shorter than the value of the ERRSTRLEN constant.

Parameters

- **err_code** – Integer error code
- **err_msg** – Pointer to put a text description.

Function sc_pipe_close2

- Defined in file_scTDC.h

Function Documentation

int **sc_pipe_close2**(const int dev_desc, const int pipe_id)

Close data pipe.

Parameters

- **dev_desc** – Device descriptor.
- **pipe_id** – Pipe id.

Returns 0 or error code.

Function sc_pipe_open2

- Defined in file_scTDC.h

Function Documentation

int **sc_pipe_open2**(const int dev_desc, const enum *sc_pipe_type_t* type, const void *params)

Open data pipe.

The user can open as many pipes as required, even if they are of the same type.

See also:

enum *sc_pipe_type_t*

See also:

struct *sc_pipe_dld_image_xy_params_t*

See also:

struct *sc_pipe_dld_image_xt_params_t*

See also:

struct *sc_pipe_dld_image_yt_params_t*

See also:

struct *sc_pipe_dld_image_3d_params_t*

See also:

struct *sc_pipe_dld_sum_histo_params_t*

See also:

struct *sc_pipe_tdc_histo_params_t*

See also:

struct *sc_pipe_statistics_params_t*

See also:

struct *sc_pipe_callback_params_t*

See also:

struct *sc_pipe_buf_callbacks_params_t*

Note: In case of internal memory allocation, if data is not read from the pipe after every exposure, the memory consumption of the application will grow. Please make sure to read data as often as necessary.

Parameters

- **dev_desc** – Device descriptor
- **type** – Pipe type.
- **params** – Pipe parameters.

Returns A non-negative (>= 0) pipe id in case of success; negative error code in case of failure. The pipe id is to be used in functions *sc_pipe_read2()* and *sc_pipe_close2()*.

Function sc_pipe_read2

- Defined in file_scTDC.h

Function Documentation

int **sc_pipe_read2**(const int dev_desc, const int pipe_id, void **buffer, unsigned int timeout)

Read data from pipe.

This function allocates memory with allocator callback from pipe parameters structure if callback function was installed or allocates memory internally. Then copies data from the last exposure and returns the memory block in buffer. If memory was allocated internally it will be deallocated when next call to the function will be performed or pipe will be closed. If memory allocation callback function was installed in the pipe parameters no any deallocation of the memory will be performed. User must manage memory by him-/her- self.

Function returns when: data is available or timeout or pipe is closed.

Parameters

- **dev_desc** – Device descriptor.
- **pipe_id** – Pipe id.
- **buffer** – Pointe to pointer where data block must be stored.
- **timeout** – Timeout in millisecond.

Function sc_tdc_cam_get_maxsize

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_maxsize**(const int dd, unsigned *width, unsigned *height)

Get width and height of the maximum possible region of interest, corresponding to the sensor size in pixels.

Parameters

- **dd** – Device Descriptor
- **width** – of the maximum possible region of interest
- **height** – of the maximum possible region of interest

Returns 0 on success, or negative error code

Function sc_tdc_cam_get_parameter

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_parameter**(const int dd, const char *name, char *value, size_t *str_len)

get the value of one of the model-specific, more specialized parameters

Parameters

- **dd** – Device Descriptor
- **name** – parameter name
- **value** – pointer to a buffer where to write the value, or nullptr
- **str_len** – if value == nullptr, the required buffer size is returned in (*str_len); if value != nullptr, (*str_len) is interpreted as the size of the buffer provided at (*value)

Returns

Function sc_tdc_cam_get_properties

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_properties**(const int dd, const int ptype, void *dest)

get properties

Parameters

- **dd** – Device Descriptor
- **ptype** – properties type
- **dest** – pointer to the cam properties data structure corresponding to the value of ptype (one of sc_CamProperties…)

Returns 0 on success, or negative error code

Function sc_tdc_cam_get_roi

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_roi**(const int dd, unsigned *x_min, unsigned *x_max, unsigned *y_min, unsigned *y_max)

Get the currently set region of interest for a camera.

Parameters

- **dd** – Device Descriptor
- **x_min** – left border
- **x_max** – right border
- **y_min** – top border
- **y_max** – bottom border

Returns 0 on success, or negative error code

Function sc_tdc_cam_get_supported_features

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_supported_features**(const int dd, unsigned *f)

Query the supported features of a camera.

Parameters

- **dd** – Device Descriptor
- **f** – filled with a bitmask that represents the supported features

Returns 0 on success, or negative error code

Function sc_tdc_cam_get_temperatures

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_get_temperatures**(const int dd, double *fpga, double *cmos)

Get the temperatures in degree Celsius of two sensors inside the housing.

Parameters

- **dd** – Device Descriptor
- **fpga** – temperature 1 (FPGA board)
- **cmos** – temperature 2 (CMOS board)

Returns 0 on success, or negative error code

Function sc_tdc_cam_set_exposure

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_set_exposure**(const int dd, unsigned e, unsigned f)

Set exposure and/or frames.

Parameters

- **dd** – Device Descriptor
- **e** – exposure in microseconds; specify zero to leave exposure unchanged
- **f** – number of frames; specify zero to leave the number of frames unchanged

Returns 0 on success, or negative error code

Function sc_tdc_cam_set_fanspeed

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_set_fanspeed**(const int dd, int fanspeed)

Set the fan speed.

Parameters

- **dd** – Device Descriptor
- **fanspeed** – accepted values range from 0 to 255 (the threshold for the fan to start rotating varies and may be somewhere between 1 and 100)

Returns 0 on success, or negative error code

Function sc_tdc_cam_set_parameter

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_set_parameter**(const int dd, const char *name, const char *value)

set one of the model-specific, more specialized parameters

Parameters **dd** – Device Descriptor

Returns 0 on success, or negative error code

Function sc_tdc_cam_set_roi

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_cam_set_roi**(const int dd, const unsigned x_min, const unsigned x_max, const unsigned y_min, const unsigned y_max)

Set a region of interest in a camera. Restriction to a subset of the sensor area may enable faster frame rates. To go back to the full sensor area, use x_min = 0, x_max = 8192, y_min = 0, y_max = 8192 (or any values for x_max, y_max that are larger than the native number of pixels on the sensor).

Parameters

- **x_min** – left border
- **dd** – Device Descriptor
- **x_max** – right border
- **y_min** – top border
- **y_max** – bottom border

Returns 0 on success, or negative error code

Function sc_tdc_config_get_key

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_config_get_key**(const int dd, const char *key, char *buf, size_t buf_size, const char *def)

Get configuration data.

Parameters

- **dd** – Device descriptor.
- **key** – Configuration key in form 'section:key'
- **buf** – Space for configuration value as a string.
- **buf_size** – Size of buf in bytes.
- **def** – Default value.

Function sc_tdc_config_get_library_version

- Defined in file_scTDC.h

Function Documentation

Warning: doxygenfunction: Unable to resolve function "sc_tdc_config_get_library_version" with arguments (unsigned) in doxygen xml output for project "scTDC library" from directory: /home/mellguth/work/surface_concept/src/cpp/sctdc1/sphinx/source/../../src/sctdc_interface_doc/xml. Potential matches:

```
- void sc_tdc_config_get_library_version(unsigned[3])
```

Function sc_tdc_deinit2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_deinit2**(const int dev_desc)

Deinitialize the hardware.

Call this function to release device and resources allocated to operate. Hardware is deinitialized after the call.

Parameters **dev_desc** – Device descriptor of the hardware to be deinitialized.

Returns int 0 or error code.

Function sc_tdc_get_binsize2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_binsize2**(const int dev_desc, double *binsize_ns)

Get the size of time bins in nanoseconds.

Parameters

- **dev_desc** – Device descriptor.
- **binsize_ns** – Pointer where binsize should be stored.

Returns 0 on success, else negative error code.

Function sc_tdc_get_blob

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_get_blob**(const int dd, char *str, size_t str_len, size_t *str_len_ret)

Get default blob algorithm.

Parameters

- **dd** – Device descriptor.
- **str** – Buffer for blob name string.
- **str_len** – Buffer for blob name string length.
- **str_len_ret** – Used memory space for blob name string.

Returns int 0 or error code.

Function sc_tdc_get_blob_parameters

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_get_blob_parameters**(const int dd, struct *sc_BlobParameters* *params)

Get blob parameters.

Function read default configuration for blob algorithm from inifile. Setting parameters with sc_tdc_set_blob_parameters does not change default parameters.

Parameters

- **dd** – Device descriptor.
- **params** – Blob parameters structure.

Returns int 0 or error code.

Function sc_tdc_get_cmos_and_smoothers_params

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_get_cmos_and_smoothers_params**(const int dd, struct *sc_CmosSmootherParameters* *params)

Get cmos and smoother parameters.

Function reads default configuration for cmos and smoother from the inifile. Setting parameters with *sc_tdc_set_cmos_and_smoothers_params()* does not change default parameters.

Parameters

- **dd** – Device descriptor.
- **params** – Cmos and smoother parameters structure.

Returns int 0 or error code.

Function sc_tdc_get_corrections2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_corrections2**(const int dd, int *ch_count, int *corrections)

get channel corrections (aka “channel shifts”)

Parameters

- **dd** – Device descriptor.
- **ch_count** – receives the number of elements needed for corrections array
- **corrections** – if nullptr, only ch_count is written to, otherwise must point to an array of ints with at least ch_count elements

Returns 0 if succesful or (negative) error code

Function sc_tdc_get_device_properties

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_device_properties**(const int dd, int params_num, void *params)

Get Device Properties. Only use for params_num <= 3.

params_num 1 coresponds to *sc_DeviceProperties1* structure. params_num 2 coresponds to *sc_DeviceProperties2* structure.

Parameters

- **dd** – Device descriptor.
- **params_num** – Number of properties structure.

- **params** – Properties structure casted to void pointer.

Function sc_tdc_get_device_properties2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_device_properties2**(const int dd, int params_num, void *params)

replacement for sc_tdc_get_device_properties when params_num > 3

Function sc_tdc_get_format2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_format2**(const int dev_desc, struct *sc_tdc_format* *format)

Get data format of the tdc events.

Call this function to get event format got from tdc pipe. See *sc_tdc_pipe_open2()* for details.

Hint: Prefer the USER_CALLBACKS pipe for development of new applications, which replaces usage of this function.

Parameters

- **dev_desc** – Device descriptor.
- **format** – Pointer on the structure where format should be placed.

Returns int 0 or error code.

Function sc_tdc_get_intens_cal_f

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_intens_cal_f**(const int dd, float *buf, size_t bufsize, size_t *size_required, unsigned *width)

Get a copy of the intensity calibration data as an image buffer.

Parameters

- **dd** – Device Descriptor
- **buf** – user provided buffer to copy to; can be 0 to get the required size
- **bufsize** – size of user provided buffer in bytes
- **size_required** – where to write the required size of the buffer in bytes
- **width** – where to write the width of the image

Returns int 0 or error code

Function sc_tdc_get_modulo2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_modulo2**(const int, unsigned int*)

Function sc_tdc_get_start_sim_params

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_start_sim_params**(const int dd, unsigned *start_count, unsigned *start_period, unsigned *start_delay, unsigned *train_count, unsigned *train_period, unsigned *start_mask)

Function sc_tdc_get_status2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_get_status2**(const int dev_desc, int *status)

Query whether the device (hardware + PC-side processing) is idle or in a measurement.

A status value 1 means the device is idle, 0 means that the device is in a measurement.

Parameters

- **dev_desc** – **[in]** Device descriptor.
- **status** – **[out]** the status value.

Returns 0 on success, else negative error code.

Function sc_tdc_init_inifile

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_init_inifile**(const char *ini_filename)

Initializes the hardware and loads the initial settings reading it from ini file.

The function must be called before any other operation and associates a non-negative integer number (device descriptor) with the device. Device descriptor must be used to identify hardware for any other operation.

Hardware with the same serial number cannot be opened twice. Call *sc_tdc_deinit2()* to close and deinitialize hardware.

Parameters `ini_filename` – Name of the inifile used for initialization.

Returns int device descriptor or error code if less then zero.

Function sc_tdc_init_inifile_override

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_init_inifile_override**(const char *ini_filename, int overrides_handle)

see *sc_tdc_init_inifile()*. Additionally copies the contents from an "override registry" and stores them internally such that values of the configuration parameters listed in that registry override those from the ini file. The registry object is not modified. Call *sc_tdc_overrides_close()* on the respective handle after initialization to avoid memory leaks unless the registry is planned for later reuse. While a similar effect can be achieved via *sc_tdc_init_with_config_lines()*, there are some differences: (1) runtime editing of the ini file in DebugLevel > 0 remains possible (2) no parsing of the ini file is required from the application developer (3) no dynamic memory allocation for data structures is required from the application developer (4) only those parameters that are to be modified need to be added into the registry

Parameters

- `ini_filename` – the name of the ini file or the full path to the ini file
- `overrides_handle` – the handle as returned by *sc_tdc_overrides_create()*. If a negative value is passed, no overrides will be in effect, and the function does NOT report an error because of this. In that case, the function has the same effect as calling *sc_tdc_init_inifile()*.

Returns non-negative device descriptor if the device was successfully initialized; else negative error code

Function sc_tdc_init_with_config_lines

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_init_with_config_lines**(const struct *sc_ConfigLine* *confline_array)

Initializes the hardware and loads the initial settings taken from array of sc_ConfigLine structures.

The function must be called before any other operation and associates a non-negative integer number (device descriptor) with the device. Device descriptor must be used to identify hardware for any other operation.

Hardware with the same serial number cannot be opened twice. Call *sc_tdc_deinit2()* to close and deinitialize hardware.

Last structure of the array must contain sc_ConfigLine::section equal NULL.

Parameters `confline_array` – Array of sc_ConfigLine structures which contains configuration needed.

Returns int device descriptor or error code if less then zero.

Function sc_tdc_interrupt2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_interrupt2**(const int dev_desc)

Interrupt a measurement before it completes.

Parameters **dev_desc** – Device descriptor.

Returns 0 on success, else negative error code.

Function sc_tdc_is_event

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_is_event**(const enum *sc_event_type_index* type, const void *event, const unsigned event_len_bytes)

Check type of event data.

When reading events via *sc_tdc_pipe_read2()*, this function can be used to test whether event is of a particular special type or not.

Returns non-zero value if event is of the type specified in the first argument, otherwise returns zero.

Deprecated:

prefer the USER_CALLBACKS pipe type

See also:

sc_pipe_open2() with *USER_CALLBACKS* from enum *sc_pipe_type_t* (defined in scTDC_types.h) as a replacement to the sc_tdc_pipe_XYZ() functions.

Parameters

- **type** – **[in]** Event type for comparison.
- **event** – **[in]** Event under testing.
- **event_len_bytes** – **[in]** Length of event in bytes.

Returns int Result of comparison.

Function sc_tdc_is_event2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_is_event2**(const enum *sc_event_type_index* type, const void *event, const unsigned event_len_bytes)

see *sc_tdc_is_event()*

Deprecated:

prefer the USER_CALLBACKS pipe type

See also:

sc_pipe_open2() with *USER_CALLBACKS* from enum *sc_pipe_type_t* (defined in scTDC_types.h) as a replacement to the sc_tdc_pipe_XYZ() functions.

Function sc_tdc_overrides_add_entry

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_overrides_add_entry**(int handle, const char *section, const char *key, const char *value)

Add an entry to an "override registry" representing a single configuration parameter whose value shall be modified. The entry consists of (1) a section and (2) a key, which correspond to those appearing in ini files, and (3) a value in string form, which will be used instead of the value in the original configuration.

Parameters

- **handle** – the non-negative handle returned from *sc_tdc_overrides_create()*
- **section** – the name of the section without the square brackets ([])
- **key** – the name of the parameter
- **value** – the overriding value in any of the string representations that would also work in ini files. However, do not use quotation marks to enclose actual string-typed parameter values.

Returns 0 on success; negative error code if the handle is unknown; the function does not check whether section, key, or value are valid.

Function sc_tdc_overrides_close

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_overrides_close**(int handle)

Delete an "override registry" and release its memory.

Parameters **handle** – the non-negative handle returned from *sc_tdc_overrides_create()*

Returns 0 on success; else negative error code (SC_TDC_ERR_NO_RESOURCE)

Function sc_tdc_overrides_create

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_overrides_create**()

Create an in-memory "override registry" to store a user-definable set of configuration parameters with values that can deviate from those in the ini file such that during initialization, the alternative values are used without any modification of the actual ini file on hard disk.

Returns a non-negative handle to an initially empty "override registry", if creation was possible; else negative error code (SC_TDC_ERR_NOMEM)

Function sc_tdc_pipe_close2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_pipe_close2**(const int dd)

Close data pipe for tdc events.

Hint: Prefer the USER_CALLBACKS pipe for development of new applications, which replaces usage of this function.

Parameters **dd** – Device descriptor.

Returns int 0 or error code.

Function sc_tdc_pipe_open2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_pipe_open2**(const int dd, size_t internal_pipe_size, const struct *sc_PipeCbf* *pipe_warning, const struct *sc_PipeCbf* *pipe_alert)

Open data pipe for tdc events.

Hint: Prefer the USER_CALLBACKS pipe for development of new applications, which replaces usage of this function.

Parameters

- **dd** – Device descriptor.
- **internal_pipe_size** – Size of internal data buffer for events in bytes.

- **pipe_warning** – 90% pipe level callback function.
- **pipe_alert** – 99% pipe level callback function.

Returns int 0 or error code.

Function sc_tdc_pipe_read2

- Defined in file_scTDC.h

Function Documentation

ssize_t **sc_tdc_pipe_read2**(const int dd, void *buffer, size_t buffer_size_bytes, unsigned timeout)

Read tdc events from the pipe.

Hint: Prefer the USER_CALLBACKS pipe for development of new applications, which replaces usage of this function.

See also:

sc_tdc_get_format2(), *sc_tdc_is_event2()*, *sc_tdc_is_event()*

Parameters

- **dd** – Device descriptor.
- **buffer** – Space for events.
- **buffer_size_bytes** – Size of the buffer in bytes.
- **timeout** – Timeout in milliseconds.

Returns ssize_t Amount of bytes were copied to the buffer or error code.

Function sc_tdc_set_blob

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_set_blob**(const int dd, const char *blob)

Set blob algorithm.

Parameters

- **dd** – Device descriptor.
- **blob** – Library name where blob algorithm implemented.

Returns int 0 or error code.

Function sc_tdc_set_blob_parameters

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_set_blob_parameters**(const int dd, const struct *sc_BlobParameters* *params)

Set blob parameters.

Parameters

- **dd** – Device descriptor.
- **params** – Blob parameters.

Returns int 0 or error code.

Function sc_tdc_set_cmos_and_smoothers_params

- Defined in file_scTDC_cam.h

Function Documentation

int **sc_tdc_set_cmos_and_smoothers_params**(const int dd, const struct *sc_CmosSmootherParameters* *params)

Set cmos and smoother parameters.

Parameters

- **dd** – Device descriptor.
- **params** – Cmos and smoother parameters structure.

Returns int 0 or error code.

Function sc_tdc_set_common_shift2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_common_shift2**(const int, const int)

Function sc_tdc_set_complete_callback2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_complete_callback2**(const int, void *owner, void (*cb)(void*, int))

Set a callback to get notifications about end of measurements.

See also:

SC_TDC_INFO_MEAS_COMPLETE, *SC_TDC_INFO_USER_INTERRUPT*, *SC_TDC_INFO_BUFFER_FULL*, *SC_TDC_INFO_HW_IDLE* (defined in scTDC_error_codes.h) that the library passes as the second argument into the cb function provided by the user.

Parameters

- **owner** – a private pointer that is replicated in the first argument of the callback
- **cb** – a callback function receiving the private pointer and a reason code.

Returns 0 on success, else negative error code.

Function sc_tdc_set_corrections2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_corrections2**(const int dd, const int ch_count, const int *corrections, const unsigned char *ch_mask)

set channel corrections (aka "channel shifts") and channel mask

Parameters

- **dd** – Device descriptor.
- **ch_count** – number of values in corrections array
- **corrections** – array of channel correction values
- **ch_mask** – array of masks with as many elements as GPX count

Returns 0 if successful or (negative) error code

Function sc_tdc_set_flim_scanner2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_flim_scanner2**(const int dd, unsigned short pixel_interval, unsigned short pixel_count, unsigned short line_count, unsigned int line_delay_interval, unsigned int multiline_count, double *corr_table)

Function sc_tdc_set_flim_scanner2ex

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_flim_scanner2ex**(const int dd, unsigned short pixel_interval, unsigned short pixel_count, unsigned short line_count, unsigned int line_delay_interval, unsigned int multiline_count, double *corr_table, unsigned int flags)

Function sc_tdc_set_iteration_number2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_iteration_number2**(const int dd, int itnum)

Set Iteration Number.

Parameters

- **dd** – Device descriptor.
- **itnum** – Number of iteration per measure.

Returns int 0 or error code.

Function sc_tdc_set_modulo2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_modulo2**(const int, const unsigned int)

Function sc_tdc_set_pulsegen_config

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_pulsegen_config**(int dev_desc, int period, int length)

Set the configuration of the pulse generator output that is available on some TDC models.

Parameters

- **dev_desc** – device descriptor
- **period** – the period of the pulse pattern in FPGA clock cycles, 0 or 1 means off
- **length** – the length of the pulse in FPGA clock cycles

Returns 0 if success, negative error code in case of failure

Function sc_tdc_set_start_sim_params

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_set_start_sim_params**(const int dd, const unsigned start_count, const unsigned start_period, const unsigned start_delay, const unsigned train_count, const unsigned train_period, const unsigned start_mask)

Set parameters for pulse train simulation generated in the TDC.

Parameters

- **dd** – Device Descriptor
- **start_count** – number of start pulses; if this number is zero, do not simulate any pulses, regardless of other parameters
- **start_period** – period of start pulses in multiples of 12.5 ns
- **start_delay** – delay of first start pulse to train pulse in multiples of 12.5 ns
- **train_count** – number of train pulses (“train triggers”); if this number is zero, do not generate train pulses, but do generate sub pulses whenever an external train pulse is detected
- **train_period** – period of train pulses in multiples of 12.5 ns
- **start_mask** – start pulse selection bitmask (a repeated concatenation of this bitmask applies to start pulses with indices larger than 32)

Returns 0 on success, or negative error code

Function sc_tdc_start_measure2

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_start_measure2**(const int dev_desc, const int ms)

Start a measurement.

Parameters

- **dev_desc** – Device descriptor.
- **ms** – Exposure time in milliseconds.

Returns 0 on success, else negative error code.

Function sc_tdc_zero_master_reset_counter

- Defined in file_scTDC.h

Function Documentation

int **sc_tdc_zero_master_reset_counter**(const int dd)

Function sc_twi_read2

- Defined in file_scTDC.h

Function Documentation

int **sc_twi_read2**(const int, const unsigned char address, unsigned char *data, const size_t size, const int stop)

Function sc_twi_set_epot2

- Defined in file_scTDC.h

Function Documentation

int **sc_twi_set_epot2**(const int, unsigned int epot, unsigned int value_number, unsigned char value)

Function sc_twi_write2

- Defined in file_scTDC.h

Function Documentation

int **sc_twi_write2**(const int, const unsigned char address, const unsigned char *data, const size_t size, const int stop)

8.4.4 Variables

Variable sc_mask32

- Defined in file_scTDC_types.h

Variable Documentation

const unsigned int **sc_mask32**[]

Used to help user to extract data fields from the event when using the sc_tdc_pipe_XYZ() functions.

Used in case of 32 bit event length

Deprecated:

use *sc_pipe_open2()* with USER_CALLBACKS from enum sc_pipe_type_t (defined in scTDC_types.h) as a replacement to the sc_tdc_pipe_XYZ() functions.

Variable sc_mask64

- Defined in file_scTDC_types.h

Variable Documentation

const unsigned long long **sc_mask64**[]

Used to help user to extract data fields from the event when using the sc_tdc_pipe_XYZ() functions.

Used in case of 64 bit event length.

Deprecated:

use *sc_pipe_open2()* with USER_CALLBACKS from enum sc_pipe_type_t (defined in scTDC_types.h) as a replacement to the sc_tdc_pipe_XYZ() functions.

8.4.5 Defines

Define ERRSTRLEN

- Defined in file_scTDC.h

Define Documentation

ERRSTRLEN 256

Define FLIM_BOTH_WAY_SCAN

- Defined in file_scTDC_types.h

Define Documentation

FLIM_BOTH_WAY_SCAN 0x01

Used in sc_tdc_set_flim_scanner2ex()

Both way scanning mode is active if this flag is on.

Define FLIM_XY_SWAP

- Defined in file_scTDC_types.h

Define Documentation

FLIM_XY_SWAP 0x02

Used in sc_tdc_set_flim_scanner2ex()

Image data are swapped if this flag is on.

Define MILLISECONDS_TO_FLOW

- Defined in file_scTDC_types.h

Define Documentation

`MILLISECONDS_TO_FLOW` 0x04

Used in *sc_tdc_format::flow_control_flags*.

Millisecond signs are placed in the tdc event stream if this flag is on.

Define SC_TDC_ERR_ALRDYINIT

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_ALRDYINIT` -16

Subsystem is already initialized. To reinitialize, deinitialize first.

Define SC_TDC_ERR_BAD_ARGUMENTS

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_BAD_ARGUMENTS` -51

Function called with bad arguments

Define SC_TDC_ERR_BADCONFI

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_BADCONFI` -9

Inifile cannot be found or has incorrect syntax.

Define SC_TDC_ERR_BIN_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_BIN_SET` -41

Define SC_TDC_ERR_BUFSIZE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_BUFSIZE` -47

Deprecated:

returned from sc_tdc_alloc_buffer() and sc_tdc_alloc_buffer_v()

Define SC_TDC_ERR_CMOS_REG

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_CMOS_REG` -95

Unable to set/get CMOS register

Define SC_TDC_ERR_CONNLOST

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_CONNLOST` -90

No connection to the device

Define SC_TDC_ERR_CORR_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_CORR_SET -40

Define SC_TDC_ERR_DEFAULT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_DEFAULT -1

General or unknown error.

Define SC_TDC_ERR_DEVCLS_INIT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_DEVCLS_INIT -14

Could not initialize device class library. Make sure that firmware file is present.

Define SC_TDC_ERR_DEVCLS_LD

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_DEVCLS_LD -12

Could not load device class library. Make sure that it is present, has correct architecture string and dependencies.

Define SC_TDC_ERR_DEVCLS_VER

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_DEVCLS_VER -13

Device class library has unsupported version. Update or use appropriate version.

Define SC_TDC_ERR_FIFO_ADDR_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FIFO_ADDR_SET -60

Unable to set fifo reading address

Define SC_TDC_ERR_FLIM_PARM_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FLIM_PARM_SET -80

Cannot set FLIM scanner parameters

Define SC_TDC_ERR_FMT_NDEF

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FMT_NDEF -50

Deprecated:

returned from sc_tdc_alloc_buffer() and sc_tdc_alloc_buffer_v()

Define SC_TDC_ERR_FMT_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FMT_SET -43

Define SC_TDC_ERR_FMT_UNSUPPORT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FMT_UNSUPPORT -44

Define SC_TDC_ERR_FPGA_INIT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_FPGA_INIT -15

Could not initialize fpga. Make sure that the firmware file is present and correct.

Define SC_TDC_ERR_GPX_FMT_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_GPX_FMT_SET -49

Could not set GPX format

Define SC_TDC_ERR_GPX_FMT_UNSUPPORT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_GPX_FMT_UNSUPPORT -48

Define SC_TDC_ERR_GPX_PLL_NLOCK

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_GPX_PLL_NLOCK -22

Could not lock PLL.

Define SC_TDC_ERR_GPX_RST

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_GPX_RST -21

Could not reset GPX. Communication with GPX broken.

Define SC_TDC_ERR_INIFILE

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_INIFILE -2

Define SC_TDC_ERR_MODE_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_MODE_SET -61

Unable to set mode

Define SC_TDC_ERR_MODULO_VALUE_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_MODULO_VALUE_SET -69

Define SC_TDC_ERR_NO_DEVICE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_DEVICE` -98

No device found

Define SC_TDC_ERR_NO_HARDWARE_SUPPORT

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_HARDWARE_SUPPORT` -9001

Feature not supported by hardware

Define SC_TDC_ERR_NO_INTENS_CAL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_INTENS_CAL` -110

No intensity calibration available

Define SC_TDC_ERR_NO_LIBRARY

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_LIBRARY` -96

Could not load library

Define SC_TDC_ERR_NO_LIBRARY_SYM

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_LIBRARY_SYM` -97

Could not find library symbol

Define SC_TDC_ERR_NO_RESOURCE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NO_RESOURCE` -99

A non-existent resource handle was specified

Define SC_TDC_ERR_NOMEM

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NOMEM` -4

System has not enough memory to perform operation.

Define SC_TDC_ERR_NOSIMFILE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NOSIMFILE` -18

No simulation data file found. Make sure that it is present and correct.

Define SC_TDC_ERR_NOT_IMPL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_NOT_IMPL` -9000

Feature not implemented

Define SC_TDC_ERR_NOTINIT

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_NOTINIT -10

Device is not initialized or bad device descriptor.

Define SC_TDC_ERR_NOTRDY

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_NOTRDY -11

Device is not ready to operate.

Define SC_TDC_ERR_OPEN_LINE_CORR_FILE

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_OPEN_LINE_CORR_FILE -81

Cannot open line_cor.txt file

Define SC_TDC_ERR_PARAMETER

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_PARAMETER -7

Define SC_TDC_ERR_POT_NO

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_POT_NO -73

Digital potentiometer is not available

Define SC_TDC_ERR_POT_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_POT_SET -74

Cannot set digital potentiometer

Define SC_TDC_ERR_ROI_BAD

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_ROI_BAD -45

Attempt to set an incorrect ROI

Define SC_TDC_ERR_ROI_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_ROI_SET -42

Setting of ROI failed

Define SC_TDC_ERR_ROI_TOOBIG

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_ROI_TOOBIG -46

Attempt to set an incorrect ROI

Define SC_TDC_ERR_SERIAL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_SERIAL` -5

Define SC_TDC_ERR_SMALLBUFFER

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_SMALLBUFFER` -8

Define SC_TDC_ERR_SPURIOUS_WAKEUP

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_SPURIOUS_WAKEUP` -19

Spurious wakeup.

Define SC_TDC_ERR_START_FAIL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_START_FAIL` -62

Could not start gpx reading

Define SC_TDC_ERR_STRT_FREQ_DIV_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_STRT_FREQ_DIV_SET` -65

Unable to set start frequency divider

Define SC_TDC_ERR_STRT_FREQ_PERIOD_GET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_STRT_FREQ_PERIOD_GET -67

Define SC_TDC_ERR_STRT_FREQ_PERIOD_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_STRT_FREQ_PERIOD_SET -66

Unable to set start frequency period

Define SC_TDC_ERR_STRT_FREQ_PERIOD_VALUE

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_STRT_FREQ_PERIOD_VALUE -68

Define SC_TDC_ERR_SYNC

- Defined in file_scTDC_error_codes.h

Define Documentation

SC_TDC_ERR_SYNC -20

Synchronisation error.

Define SC_TDC_ERR_SYSTEM

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_SYSTEM` -1000

No enough resource available

Define SC_TDC_ERR_TAG_FREQ_PERIOD_GET

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TAG_FREQ_PERIOD_GET` -120

Unable to retrieve period of tag pulses

Define SC_TDC_ERR_TDCOPEN

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TDCOPEN` -3

Define SC_TDC_ERR_TDCOPEN2

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TDCOPEN2` -6

Define SC_TDC_ERR_TIMEOUT

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TIMEOUT` -17

Timeout during reading of data.

Define SC_TDC_ERR_TIMER_EX_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TIMER_EX_SET` -64

Unable to set time range extender

Define SC_TDC_ERR_TIMER_SET

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TIMER_SET` -63

Unable to set timer

Define SC_TDC_ERR_TWI_FAIL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TWI_FAIL` -71

Unable to operate TWI module

Define SC_TDC_ERR_TWI_NACK

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TWI_NACK` -72

No acknowledge received from TWI slave

Define SC_TDC_ERR_TWI_NO_MODULE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_TWI_NO_MODULE` -70

TWI module is not available

Define SC_TDC_ERR_USB_COMM

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_USB_COMM` -30

Define SC_TDC_ERR_WRONG_LINE_CORR_FILE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_ERR_WRONG_LINE_CORR_FILE` -82

Wrong line_cor.txt file

Define SC_TDC_INFO_BUFFER_FULL

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_INFO_BUFFER_FULL` 3

Reason code used in callback registered via *sc_tdc_set_complete_callback2()*. Indicates aborted measurement because a buffer capacity was exhausted.

Define SC_TDC_INFO_HW_IDLE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_INFO_HW_IDLE` 4

Reason code used in callback registered via *sc_tdc_set_complete_callback2()*. Indicates that the hardware reached an idle state after a measurement while data is still being processed on the PC side. Typically followed by a callback with reason SC_TDC_INFO_MEAS_COMPLETE.

Define SC_TDC_INFO_MEAS_COMPLETE

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_INFO_MEAS_COMPLETE` 1

Reason code used in callback registered via *sc_tdc_set_complete_callback2()*. Indicates regular completion of the measurement.

Define SC_TDC_INFO_USER_INTERRUPT

- Defined in file_scTDC_error_codes.h

Define Documentation

`SC_TDC_INFO_USER_INTERRUPT` 2

Reason code used in callback registered via *sc_tdc_set_complete_callback2()*. Indicates measurement interrupted by user.

Define SEPARATORS_TO_FLOW

- Defined in file_scTDC_types.h

Define Documentation

`SEPARATORS_TO_FLOW` 0x01

Used in *sc_tdc_format::flow_control_flags*.

Start and Beginning of statistics is placed in the tdc event stream if this flag is on.

Define STATISTICS_TO_FLOW

- Defined in file_scTDC_types.h

Define Documentation

`STATISTICS_TO_FLOW` 0x02

Used in *sc_tdc_format::flow_control_flags*.

1024 bytes of raw statistics is placed in the endo of tdc event stream if this flag is on.

Index

B

bitsize_t (*C++ enum*), 70
bitsize_t::BS16 (*C++ enumerator*), 70
bitsize_t::BS32 (*C++ enumerator*), 70
bitsize_t::BS64 (*C++ enumerator*), 70
bitsize_t::BS8 (*C++ enumerator*), 70
bitsize_t::F32 (*C++ enumerator*), 71
bitsize_t::F64 (*C++ enumerator*), 71

E

ERRSTRLEN (*C macro*), 96

F

FLIM_BOTH_WAY_SCAN (*C macro*), 96
FLIM_XY_SWAP (*C macro*), 96

M

MILLISECONDS_TO_FLOW (*C macro*), 97

R

roi_t (*C++ struct*), 43
roi_t::offset (*C++ member*), 43
roi_t::size (*C++ member*), 43

S

sc3d_t (*C++ struct*), 44
sc3d_t::time (*C++ member*), 44
sc3d_t::x (*C++ member*), 44
sc3d_t::y (*C++ member*), 44
sc3du_t (*C++ struct*), 44
sc3du_t::time (*C++ member*), 44
sc3du_t::x (*C++ member*), 44
sc3du_t::y (*C++ member*), 44
sc_BlobParameters (*C++ struct*), 45
sc_BlobParameters::dif_min_bottom (*C++ member*), 45
sc_BlobParameters::dif_min_top (*C++ member*), 45
sc_BlobParameters::unbinning (*C++ member*), 45
sc_BlobParameters::z_scale_factor (*C++ member*), 45
sc_cam_blob_meta_t (*C++ struct*), 45
sc_cam_blob_meta_t::data_offset (*C++ member*), 45
sc_cam_blob_meta_t::nr_blobs (*C++ member*), 45
sc_cam_blob_position_t (*C++ struct*), 46
sc_cam_blob_position_t::x (*C++ member*), 46
sc_cam_blob_position_t::y (*C++ member*), 46
sc_cam_frame_meta_flags_t (*C++ enum*), 71
sc_cam_frame_meta_flags_t::SC_CAM_FRAME_HAS_IMAGE_DATA (*C++ enumerator*), 71
sc_cam_frame_meta_flags_t::SC_CAM_FRAME_IS_LAST_FRAME (*C++ enumerator*), 71
sc_cam_frame_meta_t (*C++ struct*), 46
sc_cam_frame_meta_t::adc (*C++ member*), 46
sc_cam_frame_meta_t::data_offset (*C++ member*), 46
sc_cam_frame_meta_t::flags (*C++ member*), 47
sc_cam_frame_meta_t::frame_idx (*C++ member*), 46
sc_cam_frame_meta_t::frame_time (*C++ member*), 46
sc_cam_frame_meta_t::height (*C++ member*), 46
sc_cam_frame_meta_t::pixelformat (*C++ member*), 47
sc_cam_frame_meta_t::reserved (*C++ member*), 47
sc_cam_frame_meta_t::roi_offset_x (*C++ member*), 46
sc_cam_frame_meta_t::roi_offset_y (*C++ member*), 46
sc_cam_frame_meta_t::width (*C++ member*), 46
sc_cam_pixelformat_t (*C++ enum*), 71
sc_cam_pixelformat_t::SC_CAM_PIXELFORMAT_UINT16 (*C++ enumerator*), 71
sc_cam_pixelformat_t::SC_CAM_PIXELFORMAT_UINT8 (*C++ enumerator*), 71
sc_CamProperties1 (*C++ struct*), 47
sc_CamProperties1::supports_adc (*C++ member*), 47
sc_CamProperties1::supports_blob (*C++ member*), 47
sc_CamProperties1::supports_upscaling (*C++ member*), 47
sc_CamProperties2 (*C++ struct*), 47
sc_CamProperties2::supports_convolution_mask (*C++ member*), 48
sc_CamProperties3 (*C++ struct*), 48
sc_CamProperties3::frame_max_intensity (*C++ member*), 48
sc_CamProperties3::sensor_max_intensity (*C++ member*), 48
sc_CmosSmootherParameters (*C++ struct*), 48
sc_CmosSmootherParameters::analog_gain (*C++ member*), 49
sc_CmosSmootherParameters::black_cal_offset (*C++ member*), 49
sc_CmosSmootherParameters::black_offset (*C++ member*), 49
sc_CmosSmootherParameters::digital_gain (*C++ member*), 49
sc_CmosSmootherParameters::dual_slope_us (*C++ member*), 49
sc_CmosSmootherParameters::frame_count (*C++ member*), 49
sc_CmosSmootherParameters::sc_ShutterMode (*C++ enum*), 48
sc_CmosSmootherParameters::sc_ShutterMode::EXTERNAL_START_INTERNAL_TIM (*C++ enumerator*), 48
sc_CmosSmootherParameters::sc_ShutterMode::EXTERNAL_START_INTERNAL_TIM (*C++ enumerator*), 49
sc_CmosSmootherParameters::sc_ShutterMode::FULLY_EXTERNAL (*C++ enumerator*), 48
sc_CmosSmootherParameters::sc_ShutterMode::IMMEDIATE_START_EXTERNAL_FI (*C++ enumerator*), 49

sc_CmosSmootherParameters::sc_ShutterMode::IMMEDIATE_START_INTERNAL_TIMER (*C++ enumerator*), 48
sc_CmosSmootherParameters::sc_ShutterMode::IMMEDIATE_START_INTERNAL_TIMER_MULTIPLE_SLOPES (*C++ enumerator*), 48
sc_CmosSmootherParameters::shutter_mode (*C++ member*), 49
sc_CmosSmootherParameters::single_slope_us (*C++ member*), 49
sc_CmosSmootherParameters::smoother_pixel_mask1 (*C++ member*), 49
sc_CmosSmootherParameters::smoother_pixel_mask2 (*C++ member*), 49
sc_CmosSmootherParameters::smoother_shift1 (*C++ member*), 49
sc_CmosSmootherParameters::smoother_shift2 (*C++ member*), 49
sc_CmosSmootherParameters::triple_slope_us (*C++ member*), 49
sc_CmosSmootherParameters::white_pixel_min (*C++ member*), 50
sc_ConfigLine (*C++ struct*), 50
sc_ConfigLine::key (*C++ member*), 50
sc_ConfigLine::section (*C++ member*), 50
sc_ConfigLine::value (*C++ member*), 50
sc_data_field_t (*C++ enum*), 72
sc_data_field_t::SC_DATA_FIELD_ADC (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_CHANNEL (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_DIF1 (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_DIF2 (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_MASTER_RST_COUNTER (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_SIGNAL1BIT (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_START_COUNTER (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_SUBDEVICE (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_TIME (*C++ enumerator*), 72
sc_data_field_t::SC_DATA_FIELD_TIME_TAG (*C++ enumerator*), 72
sc_DeviceProperties1 (*C++ struct*), 50
sc_DeviceProperties1::detector_size (*C++ member*), 51
sc_DeviceProperties1::pixel_size_t (*C++ member*), 51
sc_DeviceProperties1::pixel_size_x (*C++ member*), 51
sc_DeviceProperties1::pixel_size_y (*C++ member*), 51
sc_DeviceProperties2 (*C++ struct*), 51
sc_DeviceProperties2::tdc_channel_number (*C++ member*), 51
sc_DeviceProperties3 (*C++ struct*), 51
sc_DeviceProperties3::dld_event_size (*C++ member*), 51
sc_DeviceProperties3::tdc_event_size (*C++ member*), 51
sc_DeviceProperties3::user_callback_size (*C++ member*), 51
sc_DeviceProperties4 (*C++ struct*), 52
sc_DeviceProperties4::auto_modulo (*C++ member*), 52
sc_DeviceProperties4::auto_start_period (*C++ member*), 52
sc_DeviceProperties5 (*C++ struct*), 52
sc_DeviceProperties5::tag_period (*C++ member*), 52
sc_dld_device_statistics_t (*C++ struct*), 53
sc_dld_device_statistics_t::events_found (*C++ member*), 53
sc_dld_device_statistics_t::events_in_roi (*C++ member*), 53
sc_dld_device_statistics_t::events_received (*C++ member*), 53
sc_dld_device_statistics_t::reserved (*C++ member*), 53
sc_dld_set_hardware_binning (*C++ function*), 75
sc_DldEvent (*C++ struct*), 53
sc_DldEvent::adc (*C++ member*), 54
sc_DldEvent::channel (*C++ member*), 53
sc_DldEvent::dif1 (*C++ member*), 53
sc_DldEvent::dif2 (*C++ member*), 53
sc_DldEvent::master_rst_counter (*C++ member*), 53
sc_DldEvent::signal1bit (*C++ member*), 54
sc_DldEvent::start_counter (*C++ member*), 53
sc_DldEvent::subdevice (*C++ member*), 53
sc_DldEvent::sum (*C++ member*), 53
sc_DldEvent::time_tag (*C++ member*), 53
sc_event_type_index (*C++ enum*), 72
sc_event_type_index::SC_TDC_SIGN_MILLISEC (*C++ enumerator*), 72
sc_event_type_index::SC_TDC_SIGN_START (*C++ enumerator*), 72
sc_event_type_index::SC_TDC_SIGN_STAT (*C++ enumerator*), 72
sc_flim_get_counters (*C++ function*), 75
sc_flimTriggersCounters (*C++ struct*), 54
sc_flimTriggersCounters::frameTriggerCounter (*C++ member*), 54
sc_flimTriggersCounters::lineTriggerCounter (*C++ member*), 54
sc_flimTriggersCounters::pixelTriggerCounter (*C++ member*), 54
sc_get_err_msg (*C++ function*), 75
sc_Logger (*C++ struct*), 54
sc_Logger::do_log (*C++ member*), 55
sc_Logger::private_data (*C++ member*), 55
sc_LoggerFacility (*C++ enum*), 73
sc_LoggerFacility::UNUSED (*C++ enumerator*), 73
sc_mask32 (*C++ member*), 95
sc_mask64 (*C++ member*), 96
sc_pipe_buf_callback_args (*C++ struct*), 55
sc_pipe_buf_callback_args::adc (*C++ member*), 56
sc_pipe_buf_callback_args::channel (*C++ member*), 55
sc_pipe_buf_callback_args::data_len (*C++ member*), 56
sc_pipe_buf_callback_args::dif1 (*C++ member*), 56
sc_pipe_buf_callback_args::dif2 (*C++ member*), 56
sc_pipe_buf_callback_args::event_index (*C++ member*), 55
sc_pipe_buf_callback_args::master_rst_counter (*C++ member*), 56
sc_pipe_buf_callback_args::ms_indices (*C++ member*), 55
sc_pipe_buf_callback_args::ms_indices_len (*C++ member*), 56
sc_pipe_buf_callback_args::reserved (*C++ member*), 56
sc_pipe_buf_callback_args::signal1bit (*C++ member*), 56
sc_pipe_buf_callback_args::som_indices (*C++ member*), 55
sc_pipe_buf_callback_args::som_indices_len (*C++ member*), 56
sc_pipe_buf_callback_args::start_counter (*C++ member*), 55
sc_pipe_buf_callback_args::subdevice (*C++ member*), 55
sc_pipe_buf_callback_args::time (*C++ member*), 56
sc_pipe_buf_callback_args::time_tag (*C++ member*), 55
sc_pipe_buf_callbacks_params_t (*C++ struct*), 56
sc_pipe_buf_callbacks_params_t::data (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::data_field_selection (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::dld_events (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::end_of_measurement (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::max_buffered_data_len (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::priv (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::reserved (*C++ member*), 57
sc_pipe_buf_callbacks_params_t::version (*C++ member*), 57
sc_pipe_callback_params_t (*C++ struct*), 57
sc_pipe_callback_params_t::callbacks (*C++ member*), 58
sc_pipe_callbacks (*C++ struct*), 58
sc_pipe_callbacks::dld_event (*C++ member*), 59

sc_pipe_callbacks::end_of_measure (*C++ member*), 58
sc_pipe_callbacks::millisecond_countup (*C++ member*), 58
sc_pipe_callbacks::priv (*C++ member*), 58
sc_pipe_callbacks::start_of_measure (*C++ member*), 58
sc_pipe_callbacks::statistics (*C++ member*), 58
sc_pipe_callbacks::tdc_event (*C++ member*), 58
sc_pipe_close2 (*C++ function*), 75
sc_pipe_dld_image_3d_params_t (*C++ struct*), 59
sc_pipe_dld_image_3d_params_t::accumulation_ms (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::allocator_cb (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::allocator_owner (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::binning (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::channel (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::depth (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::modulo (*C++ member*), 59
sc_pipe_dld_image_3d_params_t::roi (*C++ member*), 59
sc_pipe_dld_image_xt_params_t (*C++ struct*), 60
sc_pipe_dld_image_xt_params_t::accumulation_ms (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::allocator_cb (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::allocator_owner (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::binning (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::channel (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::depth (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::modulo (*C++ member*), 60
sc_pipe_dld_image_xt_params_t::roi (*C++ member*), 60
sc_pipe_dld_image_xy_ext_params_t (*C++ struct*), 61
sc_pipe_dld_image_xy_ext_params_t::base (*C++ member*), 61
sc_pipe_dld_image_xy_ext_params_t::extension (*C++ member*), 61
sc_pipe_dld_image_xy_params_t (*C++ struct*), 61
sc_pipe_dld_image_xy_params_t::accumulation_ms (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::allocator_cb (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::allocator_owner (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::binning (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::channel (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::depth (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::modulo (*C++ member*), 61
sc_pipe_dld_image_xy_params_t::roi (*C++ member*), 61
sc_pipe_dld_image_yt_params_t (*C++ struct*), 62
sc_pipe_dld_image_yt_params_t::accumulation_ms (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::allocator_cb (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::allocator_owner (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::binning (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::channel (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::depth (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::modulo (*C++ member*), 62
sc_pipe_dld_image_yt_params_t::roi (*C++ member*), 62
sc_pipe_dld_stat_params_t (*C++ struct*), 63
sc_pipe_dld_stat_params_t::allocator_cb (*C++ member*), 63
sc_pipe_dld_stat_params_t::allocator_owner (*C++ member*), 63
sc_pipe_dld_stat_params_t::device_number (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t (*C++ struct*), 63
sc_pipe_dld_sum_histo_params_t::accumulation_ms (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::allocator_cb (*C++ member*), 64
sc_pipe_dld_sum_histo_params_t::allocator_owner (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::binning (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::channel (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::depth (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::modulo (*C++ member*), 63
sc_pipe_dld_sum_histo_params_t::roi (*C++ member*), 63
sc_pipe_image_source (*C++ struct*), 64
sc_pipe_image_source::extension (*C++ member*), 64
sc_pipe_image_source::sc_ImageSource (*C++ enum*), 64
sc_pipe_image_source::sc_ImageSource::BOTH (*C++ enumerator*), 64
sc_pipe_image_source::sc_ImageSource::EVENTS (*C++ enumerator*), 64
sc_pipe_image_source::sc_ImageSource::RAMDATA (*C++ enumerator*), 64
sc_pipe_image_source::source (*C++ member*), 64
sc_pipe_image_source::type (*C++ member*), 64
sc_pipe_open2 (*C++ function*), 76
sc_pipe_param_extension_type (*C++ enum*), 73
sc_pipe_param_extension_type::SC_PIPE_PARAM_EXTENSION_TYPE_IMAGE_SOURC (*C++ enumerator*), 73
sc_pipe_read2 (*C++ function*), 77
sc_pipe_statistics_params_t (*C++ struct*), 65
sc_pipe_statistics_params_t::allocator_cb (*C++ member*), 65
sc_pipe_statistics_params_t::allocator_owner (*C++ member*), 65
sc_pipe_tdc_histo_params_t (*C++ struct*), 65
sc_pipe_tdc_histo_params_t::accumulation_ms (*C++ member*), 65
sc_pipe_tdc_histo_params_t::allocator_cb (*C++ member*), 66
sc_pipe_tdc_histo_params_t::allocator_owner (*C++ member*), 66
sc_pipe_tdc_histo_params_t::binning (*C++ member*), 65
sc_pipe_tdc_histo_params_t::channel (*C++ member*), 65
sc_pipe_tdc_histo_params_t::depth (*C++ member*), 65
sc_pipe_tdc_histo_params_t::modulo (*C++ member*), 65
sc_pipe_tdc_histo_params_t::offset (*C++ member*), 65
sc_pipe_tdc_histo_params_t::size (*C++ member*), 65
sc_pipe_tdc_stat_params_t (*C++ struct*), 66
sc_pipe_tdc_stat_params_t::allocator_cb (*C++ member*), 66
sc_pipe_tdc_stat_params_t::allocator_owner (*C++ member*), 66
sc_pipe_tdc_stat_params_t::channel_number (*C++ member*), 66
sc_pipe_type_t (*C++ enum*), 73
sc_pipe_type_t::BUFFERED_DATA_CALLBACKS (*C++ enumerator*), 74
sc_pipe_type_t::DLD_IMAGE_3D (*C++ enumerator*), 74
sc_pipe_type_t::DLD_IMAGE_XT (*C++ enumerator*), 73
sc_pipe_type_t::DLD_IMAGE_XY (*C++ enumerator*), 73
sc_pipe_type_t::DLD_IMAGE_XY_EXT (*C++ enumerator*), 74
sc_pipe_type_t::DLD_IMAGE_YT (*C++ enumerator*), 74
sc_pipe_type_t::DLD_STATISTICS (*C++ enumerator*), 74
sc_pipe_type_t::DLD_SUM_HISTO (*C++ enumerator*), 74
sc_pipe_type_t::PIPE_CAM_BLOBS (*C++ enumerator*), 74
sc_pipe_type_t::PIPE_CAM_FRAMES (*C++ enumerator*), 74
sc_pipe_type_t::STATISTICS (*C++ enumerator*), 74

sc_pipe_type_t::TDC_HISTO (*C++ enumerator*), 73
sc_pipe_type_t::TDC_STATISTICS (*C++ enumerator*), 74
sc_pipe_type_t::TMSTAMP_TDC_HISTO (*C++ enumerator*), 74
sc_pipe_type_t::USED_MEM_CALLBACKS (*C++ enumerator*), 74
sc_pipe_type_t::USER_CALLBACKS (*C++ enumerator*), 74
sc_pipe_used_mem_callbacks_params_t (*C++ struct*), 66
sc_pipe_used_mem_callbacks_params_t::priv (*C++ member*), 66
sc_pipe_used_mem_callbacks_params_t::used_mem (*C++ member*), 66
sc_PipeCbf (*C++ struct*), 67
sc_PipeCbf::cb (*C++ member*), 67
sc_PipeCbf::private_data (*C++ member*), 67
sc_tdc_cam_get_maxsize (*C++ function*), 77
sc_tdc_cam_get_parameter (*C++ function*), 78
sc_tdc_cam_get_properties (*C++ function*), 78
sc_tdc_cam_get_roi (*C++ function*), 78
sc_tdc_cam_get_supported_features (*C++ function*), 79
sc_tdc_cam_get_temperatures (*C++ function*), 79
sc_tdc_cam_set_exposure (*C++ function*), 79
sc_tdc_cam_set_fanspeed (*C++ function*), 80
sc_tdc_cam_set_parameter (*C++ function*), 80
sc_tdc_cam_set_roi (*C++ function*), 80
sc_tdc_channel_statistics_t (*C++ struct*), 67
sc_tdc_channel_statistics_t::counter (*C++ member*), 67
sc_tdc_channel_statistics_t::counts_read (*C++ member*), 67
sc_tdc_channel_statistics_t::counts_received (*C++ member*), 67
sc_tdc_channel_statistics_t::reserved (*C++ member*), 67
sc_tdc_config_get_key (*C++ function*), 81
sc_tdc_deinit2 (*C++ function*), 81
SC_TDC_ERR_ALRDYINIT (*C macro*), 97
SC_TDC_ERR_BAD_ARGUMENTS (*C macro*), 97
SC_TDC_ERR_BADCONFI (*C macro*), 97
SC_TDC_ERR_BIN_SET (*C macro*), 98
SC_TDC_ERR_BUFSIZE (*C macro*), 98
SC_TDC_ERR_CMOS_REG (*C macro*), 98
SC_TDC_ERR_CONNLOST (*C macro*), 98
SC_TDC_ERR_CORR_SET (*C macro*), 99
SC_TDC_ERR_DEFAULT (*C macro*), 99
SC_TDC_ERR_DEVCLS_INIT (*C macro*), 99
SC_TDC_ERR_DEVCLS_LD (*C macro*), 99
SC_TDC_ERR_DEVCLS_VER (*C macro*), 99
SC_TDC_ERR_FIFO_ADDR_SET (*C macro*), 100
SC_TDC_ERR_FLIM_PARM_SET (*C macro*), 100
SC_TDC_ERR_FMT_NDEF (*C macro*), 100
SC_TDC_ERR_FMT_SET (*C macro*), 100
SC_TDC_ERR_FMT_UNSUPPORT (*C macro*), 101
SC_TDC_ERR_FPGA_INIT (*C macro*), 101
SC_TDC_ERR_GPX_FMT_SET (*C macro*), 101
SC_TDC_ERR_GPX_FMT_UNSUPPORT (*C macro*), 101
SC_TDC_ERR_GPX_PLL_NLOCK (*C macro*), 101
SC_TDC_ERR_GPX_RST (*C macro*), 102
SC_TDC_ERR_INIFILE (*C macro*), 102
SC_TDC_ERR_MODE_SET (*C macro*), 102
SC_TDC_ERR_MODULO_VALUE_SET (*C macro*), 102
SC_TDC_ERR_NO_DEVICE (*C macro*), 103
SC_TDC_ERR_NO_HARDWARE_SUPPORT (*C macro*), 103
SC_TDC_ERR_NO_INTENS_CAL (*C macro*), 103
SC_TDC_ERR_NO_LIBRARY (*C macro*), 103
SC_TDC_ERR_NO_LIBRARY_SYM (*C macro*), 103
SC_TDC_ERR_NO_RESOURCE (*C macro*), 104
SC_TDC_ERR_NOMEM (*C macro*), 104
SC_TDC_ERR_NOSIMFILE (*C macro*), 104
SC_TDC_ERR_NOT_IMPL (*C macro*), 104
SC_TDC_ERR_NOTINIT (*C macro*), 105
SC_TDC_ERR_NOTRDY (*C macro*), 105
SC_TDC_ERR_OPEN_LINE_CORR_FILE (*C macro*), 105
SC_TDC_ERR_PARAMETER (*C macro*), 105
SC_TDC_ERR_POT_NO (*C macro*), 105
SC_TDC_ERR_POT_SET (*C macro*), 106
SC_TDC_ERR_ROI_BAD (*C macro*), 106
SC_TDC_ERR_ROI_SET (*C macro*), 106
SC_TDC_ERR_ROI_TOOBIG (*C macro*), 106
SC_TDC_ERR_SERIAL (*C macro*), 107
SC_TDC_ERR_SMALLBUFFER (*C macro*), 107
SC_TDC_ERR_SPURIOUS_WAKEUP (*C macro*), 107
SC_TDC_ERR_START_FAIL (*C macro*), 107
SC_TDC_ERR_STRT_FREQ_DIV_SET (*C macro*), 107
SC_TDC_ERR_STRT_FREQ_PERIOD_GET (*C macro*), 108
SC_TDC_ERR_STRT_FREQ_PERIOD_SET (*C macro*), 108
SC_TDC_ERR_STRT_FREQ_PERIOD_VALUE (*C macro*), 108
SC_TDC_ERR_SYNC (*C macro*), 108
SC_TDC_ERR_SYSTEM (*C macro*), 109
SC_TDC_ERR_TAG_FREQ_PERIOD_GET (*C macro*), 109
SC_TDC_ERR_TDCOPEN (*C macro*), 109
SC_TDC_ERR_TDCOPEN2 (*C macro*), 109
SC_TDC_ERR_TIMEOUT (*C macro*), 109
SC_TDC_ERR_TIMER_EX_SET (*C macro*), 110
SC_TDC_ERR_TIMER_SET (*C macro*), 110
SC_TDC_ERR_TWI_FAIL (*C macro*), 110
SC_TDC_ERR_TWI_NACK (*C macro*), 110
SC_TDC_ERR_TWI_NO_MODULE (*C macro*), 111
SC_TDC_ERR_USB_COMM (*C macro*), 111
SC_TDC_ERR_WRONG_LINE_CORR_FILE (*C macro*), 111
sc_tdc_format (*C++ struct*), 67
sc_tdc_format::channel_length (*C++ member*), 68
sc_tdc_format::channel_offset (*C++ member*), 68
sc_tdc_format::dif1_length (*C++ member*), 68
sc_tdc_format::dif1_offset (*C++ member*), 68
sc_tdc_format::dif2_length (*C++ member*), 68
sc_tdc_format::dif2_offset (*C++ member*), 68
sc_tdc_format::flow_control_flags (*C++ member*), 69
sc_tdc_format::reserved (*C++ member*), 69
sc_tdc_format::sign_counter_length (*C++ member*), 68
sc_tdc_format::sign_counter_offset (*C++ member*), 68
sc_tdc_format::start_counter_length (*C++ member*), 68
sc_tdc_format::start_counter_offset (*C++ member*), 68
sc_tdc_format::sum_length (*C++ member*), 68
sc_tdc_format::sum_offset (*C++ member*), 68
sc_tdc_format::time_data_length (*C++ member*), 68
sc_tdc_format::time_data_offset (*C++ member*), 68
sc_tdc_format::time_tag_length (*C++ member*), 68
sc_tdc_format::time_tag_offset (*C++ member*), 68
sc_tdc_format::total_bits_length (*C++ member*), 68
sc_tdc_get_binsize2 (*C++ function*), 82
sc_tdc_get_blob (*C++ function*), 82
sc_tdc_get_blob_parameters (*C++ function*), 82
sc_tdc_get_cmos_and_smoothers_params (*C++ function*), 83
sc_tdc_get_corrections2 (*C++ function*), 83
sc_tdc_get_device_properties (*C++ function*), 83
sc_tdc_get_device_properties2 (*C++ function*), 84
sc_tdc_get_format2 (*C++ function*), 84
sc_tdc_get_intens_cal_f (*C++ function*), 84
sc_tdc_get_modulo2 (*C++ function*), 85
sc_tdc_get_start_sim_params (*C++ function*), 85
sc_tdc_get_status2 (*C++ function*), 85
SC_TDC_INFO_BUFFER_FULL (*C macro*), 111
SC_TDC_INFO_HW_IDLE (*C macro*), 111
SC_TDC_INFO_MEAS_COMPLETE (*C macro*), 112
SC_TDC_INFO_USER_INTERRUPT (*C macro*), 112
sc_tdc_init_inifile (*C++ function*), 85
sc_tdc_init_inifile_override (*C++ function*), 86
sc_tdc_init_with_config_lines (*C++ function*), 86
sc_tdc_interrupt2 (*C++ function*), 87
sc_tdc_is_event (*C++ function*), 87
sc_tdc_is_event2 (*C++ function*), 88
sc_tdc_overrides_add_entry (*C++ function*), 88
sc_tdc_overrides_close (*C++ function*), 89
sc_tdc_overrides_create (*C++ function*), 89
sc_tdc_pipe_close2 (*C++ function*), 89
sc_tdc_pipe_open2 (*C++ function*), 89
sc_tdc_pipe_read2 (*C++ function*), 90
sc_tdc_set_blob (*C++ function*), 90
sc_tdc_set_blob_parameters (*C++ function*), 91

sc_tdc_set_cmos_and_smoothers_params (*C++ function*), 91
sc_tdc_set_common_shift2 (*C++ function*), 91
sc_tdc_set_complete_callback2 (*C++ function*), 92
sc_tdc_set_corrections2 (*C++ function*), 92
sc_tdc_set_flim_scanner2 (*C++ function*), 92
sc_tdc_set_flim_scanner2ex (*C++ function*), 93
sc_tdc_set_iteration_number2 (*C++ function*), 93
sc_tdc_set_modulo2 (*C++ function*), 93
sc_tdc_set_pulsegen_config (*C++ function*), 93
sc_tdc_set_start_sim_params (*C++ function*), 94
sc_tdc_start_measure2 (*C++ function*), 94
sc_tdc_zero_master_reset_counter (*C++ function*), 95
sc_TdcEvent (*C++ struct*), 69
sc_TdcEvent::channel (*C++ member*), 69
sc_TdcEvent::sign_counter (*C++ member*), 69
sc_TdcEvent::start_counter (*C++ member*), 69
sc_TdcEvent::subdevice (*C++ member*), 69
sc_TdcEvent::time_data (*C++ member*), 69
sc_TdcEvent::time_tag (*C++ member*), 69
sc_twi_read2 (*C++ function*), 95
sc_twi_set_epot2 (*C++ function*), 95
sc_twi_write2 (*C++ function*), 95
SEPARATORS_TO_FLOW (*C macro*), 112
statistics_t (*C++ struct*), 70
statistics_t::counters (*C++ member*), 70
statistics_t::counts_read (*C++ member*), 70
statistics_t::counts_received (*C++ member*), 70
statistics_t::events_found (*C++ member*), 70
statistics_t::events_in_roi (*C++ member*), 70
statistics_t::events_received (*C++ member*), 70
statistics_t::reserved (*C++ member*), 70
STATISTICS_TO_FLOW (*C macro*), 112