scTDCLibrary

scTDC library interface

Version

1.3000.0

This is documentation for scTDC library API. scTDC library is used for access to Surface Concept GmbH Time to Digital Converter and Delay Line Detector devices.

Intro

Starting from the version 1.4.0 of scTDC library new abstraction was implemented to extract tdc and dld data from the device. The main mechanism for that is something which is called data pipe. Data pipes can be of different type and have different parameter. Every data pipe get pipe ident number when is opened with **sc_pipe_open2()** function. There could be as many pipes opened as application requires. After measure is started with sc_start_measure2() application should read the data from the data pipe with the **sc_pipe_read2()** function. When operation is finished data pipe should be closed with **sc_pipe_close2()**.

Here is minimal example how things should be done:

```
//c99 compiler must be used
#include <scTDC.h>
#include <stdio.h>

int main()
{
  int dd = sc_tdc_init_inifile("tdc_gpx3.ini");
  if (dd < 0) { // could not initialise 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 could be used for any other operation
    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 lets try to open 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

    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 1000 ms measure

    unsigned *tdc_histo;
    ret = sc_pipe_read2(dd, pipe_id, (void *)&tdc_histo, -1); //after the call
    //tdc_histo pointer will point on the histogram data. The place for the
    //histogram will be allocated by internal allocator and will be destroyed
    //at next call. -1 means 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 data, may process it or show somewhere

    sc_pipe_close2(dd, pipe_id);

    //here data pipe is closed. NOTE: tdc_histo now points to nowhere. If
    //application needs tdc_data at this point - copy it before calling
    //sc_pipe_close2().

    sc_tdc_deinit2(dd); // release hardware and resources.
```

```
  return 0;
 }
}
```

Configuration

To configure scTDC1 library application must supply inifile and firmware file (both supplied with the device). Inifile contains information about firmware file which will be used. That's why sc_tdc_init_inifile2() call takes only one parameter which is the name of the inifile. sc_tdc_init_inifile2() call returns positive integer number which serves as a device descriptor and must be user in all other calls related to the configured device. Negative returned value means error. There can be text description of the error obtained with the **sc_get_err_msg()** call.

Start Measure

sc_tdc_start_measure2() is used to start measure procedure. After the call device goes to the measurement state, extracts data from the device analizes it and transfers to the application through data pipes which must be configured previously. See **Extracting Data** section for more info how to operate with data pipes. In case of external start **sc_tdc_start_measure2()** call only transfer scTDC1 to the state of waiting for the start pulse on the device input. Currently external start feature must be switched in the inifile. Unfortunately there is no way to do that through scTDC API but this may be changed in future.

Extracting Data

The main abstraction unit of the scTDC1 API which intend to be used for extracting processed (or raw) data from the library is a data pipe. Application can open as many data pipes as required for operation. All of them can have their own parameters, settings and types. The only limitation is the machine resources like memory and processor power. Due to of historical and optimisation reasons the processing happens in only one thread. This means that amount of time required to process one data unit (some number of tdc events) is linearly growing up with number of data pipes opened. The data processing mechanism may be changed in the future to be multithread.

Data pipe can be opened with **sc_pipe_open2()** call. Data can be extracted with **sc_pipe_read2()** call. **sc_pipe_close2()** call is used to close pipe and free resources used.

Here is a little example how data pipe for 2d images can be opened and operated (2d images available only when using dld device).

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

 int dd = sc_tdc_init_inifile("tdc_gpx3.ini"); //init the hardware and get device descriptor
 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}};

 int pd = sc_pipe_open2(dd, DLD_IMAGE_XY, (void *)&prms); //init image 2d pipe and get pipe descriptor
 //can be called several times with different parameters if necessary
 if (pd < 0) {
  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 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_deinit2(dd);
  return ret;
 }

 uint32_t *image;
 //This will unblock when exposure finished. image will be set to internally
 // allocated image data. Size of the image is roi_x  * roi_y * 4
 // Here it will be 512 * 512 * 4 (see prms.roi and prms.depth settings).
 // Deallocation happens when next sc_pipe_read2(), sc_pipe_close2() or
 //sc_tdc_deinit2() call.
 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 application should do necessary actions with the image.
  for (size_t i=0; i<size_x * size_y; ++i) {
    fprintf(stderr, "%08x\n", image[i]);
  }

  sc_pipe_close2(dd, pd);
  sc_tdc_deinit2(dd);
  fprintf(stderr, "\n");
  return 0;
}
```

Data Pipe Memory Usage Notes

Due to of non-automatic memory handling in C programming language the question of allocation and deallocation memory is very important. Currently scTDC1 has two ways of memory treatment for data pipes. One is so called 'internal' memory allocation, when memory is allocated by scTDC1 and deallocated in the moment when next **sc_pipe_read2()**, **sc_pipe_close2()** or **sc_tdc_deinit2()** called. Second one - 'external' - when application supplies allocator callback function in data pipe parameters. In this case allocator callback function is called every time when data pipe processing algorithm needs memory for the data. Deallocation must be performed by the application.

Simple example for statistics pipe with external allocator:

```
class Allocator
{
    // This is allocator class which will store 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_); //scTDC does not 'zeroing' memory supplied!
      *u = &(chunk[0]);
      mem_chunks_.push_back(std::move(chunk));
      return 0;
    }
};

int statistics_pipe_ex()
{
  int dd = sc_tdc_init_inifile("tdc_gpx3.ini"); //init the hardware and get device descriptor
  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) {
    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 measure processing engine will call mem.pre_alloc function, which
  //allocates 1k chunk of memory, save it in the list and return to the processor.
  //The next function will unblock after measure is finished. Application
  //will get back statistics in the memory, which will be deallocated when mem
  //object is destroyed. If application call sc_start_measure2 & sc_pipe_read2()
  //many times here mem object will accumulate memory chunks in the list.
  //Other nice application of this feature is making accumulation of the data
  // in the same memory space by giving always the same pointer. scTDC will not
  //'zeroed' memory supplied and data will accumulate in the same place.
  //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 statistics

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

  //Here statistics is still accessible because mem object is still on the stack.
  //If 'internal' memory allocation used this would be not true.
  return 0;
}
```

User Callback Interface

Since version 1.3000.0 there was new data extraction mechanism implemented. The machanism uses set of callback functions provided by user to notify about one or another event or data delieved from the device.

Here is simple example:

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

struct sc_DeviceProperties3 sizes;

/* Due to of different implementation of semaphores in different operation
   systems concrete implementation is not provided here */
typedef int Semaphore;

 /* As mentioned above next function does nothing currently and must be
   implemented differently for different operation systems */
void sem_inc(Semaphore *a) {}
void sem_dec(Semaphore *a) {}

struct PrivData {
  Semaphore sem;
};

void cb_start(void *p) {
  /* this function will be called when measurement is started */
}

void cb_end(void *p) {
  /* this function will be called when measurement is finished */
  struct PrivData *d = (struct PrivData *)p;
  sem_inc(&d->sem);
}

void cb_millis(void *p) {
  /* this function will be called when millisecond tick received */
}

void cb_stat(void *p, const struct statistics_t *stat) {
  /* this function will be called when measurement statistics received */
}

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);
    /* there must be some code doing something with tdc event which
      represeted as obj pointer.
      obj->channel, obj->time_data ... contain information about
      tdc event received */
  }
}

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);
    /* there must be some code doing something with dld event which
      represeted as obj pointer.
      obj->dif1, obj->dif2, obj->sum ... contain information about
      dld event received */
  }
}


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;
  }

  /* Please note that size of sc_pipe_callbacks, sc_TdcEvent or sc_DldEvent
  structures may be changed in next versions of api. This code example
  written in such a way that size changes will not do any incorrect code
  behaviour in such case */
  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;
  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;
  }

  /* In this way we wait untill measurement is finished */
  sem_dec(&priv_data.sem);

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

  return 0;
}
```

Old API Notes

Function which is marked as DEPRECATED belongs to the old API which has a number of problems which in principle cannot be resolved. It is still supported to make old applications work, but will be removed in the future. Please refrain from using it in new applications.