quTAG

Time-to-Digital Converter

Software manual V1.5.0

2019-11-20

qutools

Contents

1 Introduction 3

2 Code examples 4

  2.1 How to handle the quTAG DLL - Example by the Python wrapper 4

  2.2 Get event and coincidence rates - Example Python / DLL functions 6

  2.3 Get timestamps and check data loss - Example Python / DLL functions 8

  2.4 Write timestamps to file - Example Python / DLL functions 9

  2.5 Create and get histograms - Example Python / DLL functions 10

  2.6 Handle multiple devices - Example Python / DLL functions 12

3 Command Line Interface 14

4 FAQ 15

  4.1 USB Driver Version 15

  4.2 Firmware Version 15

  4.3 Unexpected Delays 15

  4.4 Low Resolution with Jitter Upgrade 16

1 Introduction

The quTAG software can be installed on Windows 7, 8, 8.1, and 10, 32- and 64 Bit. The Linux version requires a x86-64 Bit distribution with libc 2.19 or newer.

On Windows, using the installer Daisy@QUTAG-Vx.y.z.exe	Start the installer program and follow the instructions. Drivers and necessary libraries will be installed automatically.
On Windows, using the zip archive QUTAG_Vx.y.z.zip	Extract the zip archive to a directory of your choice. Install the device driver in the usbdriver directory using dpinst32.exe or dpinst64.exe, whatever conforms to your Windows version.
On Linux, using the tgz archive QUTAG_Vx.y.z.tgz	Extract the archive to a directory of your choice. Follow the instructions in install/readme.linux.txt.

The following software will now reside in the installation directory.

daisy(.exe)	"Data Analysis and Imaging System" – the main control software for the quTAG.
nhflash(.exe)	The firmware update tool.
tarec(.exe)	A record & merge tool for synchronized devices.
tdccli(.exe)	A command line tool for simple tasks.
userlib	The directory contains the custom programming library, with HTML documentation and LabView wrapper Vis.
usbdriver	USB driver packet for Windows 32 and 64 Bit.
firmware	Firmware for the device.

See the quTAG manual for more information.

When using the Windows installer, also start menu entries for the programs are created.

2 Code examples

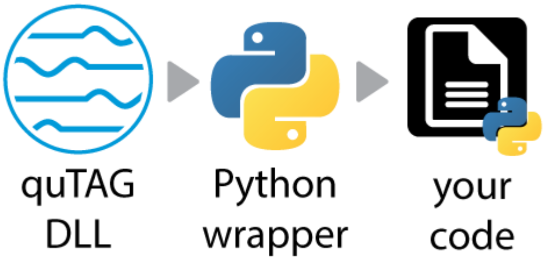
2.1 How to handle the quTAG DLL - Example by the Python wrapper

This chapter explains the handling of the DLL by the python wrapper and help the understanding how the initialization and communication with the quTAG works in general.

What you need to start:

- The wrapper works with **python 3.x**.
- If you work with **python 32 Bit**, target the 32 Bit DLL **`tdcbase.dll`** – downloaded with the standard quTAG software in the Downloads tab at https://www.qutools.com/qutag/
- If you work with **python 64 Bit**, target the 64 Bit DLL **`tdcbase.dll`** – the library can be downloaded as a ZIP file in the Downloads tab under Software -> 64bit library.

If you want a quick start, skip this chapter and get right in the first example.



The python examples are based on the python wrapper **`QuTAG.py`**, which loads the quTAG DLL **`tdcbase.dll`** and wraps all functions for python and initializes some parameters automatically.

The wrapper first imports `ctypes` to handle the DLL and `os` to change the working directory automatically, see below.

```
import ctypes
import os
```

The wrapper **`QuTAG.py`** and all your individual codes should be located in the software path "\QUTAG-Vx.x.x\userlib\src\" to use the **`tdcbase.dll`** in "\QUTAG-Vx.x.x\userlib\lib\". If you want to change to individual directories edit the DLL search path within **`QuTAG.py`** wrapper in the initialization.

```
class QuTAG:

    def __init__(self):

        self.qutools_dll = ctypes.windll.LoadLibrary(dll_name)
```

before `LoadLibrary`, the working directory is changed to the wrapper path.

Then the API of the DLL is declared. Here all functions of the DLL are defined and wrapped to use with python by `ctypes`. You can look up the functions in the documentation in "\QUTAG-Vx.x.x\userlib\documentation.html".

```
156.            self.qutools_dll.TDC_deInit.argtypes = None
157.            self.qutools_dll.TDC_deInit.restype = ctypes.c_int32
```

And some dictionaries for parameters and errors are implemented.

After that, the actual quTAG device is initialized within the wrapper.

```
58.             self.Initialize()
```

```
372. # Init ---------------------------------------------------------------
373.        def Initialize(self):
374.            ans = self.qutools_dll.TDC_init(self.dev_nr)
```

> The wrapped DLL function `TDC_init` is called. See also the documentation in "\QUTAG-Vx.x.x\userlib\documentation.html" for this and the following DLL comments.

The function `TDC_init` from the DLL and wrapped in the APT declaration disconnects from any connected devices, initializes internal data and starts an event loop for data acquisition. It discovers devices connected to the computer, and connects to the first device that matches the given device ID. In the `__init__(self)` function of the wrapper (line 41), a device ID -1 is defined before the function `Initialize` is called. The device ID is an identification number programmed by the user. The special value -1 matches all devices.

```
56.         self.dev_nr=-1
```

The function `TDC_init` **should be called before any other TDC functions**. Alternatively, `TDC_discover` and `TDC_connect` can be used to handle multiple devices from one program.

If no device is found, a demo mode is started. So, you can test your code without hardware. The wrapper will tell you by printing "**No suitable device found - demo mode activated**" in the python shell. But if a device is connected, you probably should reconnect the USB connection or restart the device.

2.2 Get event and coincidence rates - Example Python / DLL functions

In the first example `qutag-GetCoincCounter-starter_example.py` we are retrieving event rates and coincidences of the quTAG and its input channels.

At first - in every example - the wrapper is imported.

```python
try:
	import QuTAG
except:
	print("Time Tagger wrapper QuTAG.py is not in the search path.")
```

If you get the error message, please read chapter 2.1 where your files should be located in respect to the wrapper and the DLL.

After that we initialize the quTAG with the following line.

```python
# Initialize the quTAG device
qutag = QuTAG.QuTAG()
```

For more info about what's happening in the initialization, see chapter 2.1 .

Then we are setting the exposure time in milliseconds for how long the quTAG should look on the channel and count the events and calculates the coincidences between channels.

```python
# Set the exposure time (or integration time) of the internal coincidence counters in milliseconds, range = 0...65535
qutag.setExposureTime(100) # 100 ms exposure time
```

> The wrapped DLL function `TDC_setExposureTime` is called. See also the documentation in "\QUTAG-Vx.x.x\userlib\documentation.html" for this and the following DLL comments.

After that we give the quTAG some time to accumulate the data with `time.sleep(1).` Therefore, in the first lines of the sample code, we imported `import time`.

Now let's retrieve the most recent values of the built-in coincidence counters from quTAG.

```python
data,updates = qutag.getCoincCounters()
```

> The wrapped DLL function `TDC_getCoincCounters` is called.

`data` is a 1D numpy array with 31 entries of int32. The array contains count rates for all 5 channels and rates for coincidences of events detected on different channels.

Events are coincident if they happen within the coincidence window. This window can be changed by the function `qutag.setCoincidenceWindow(int32)`. The parameter for the coincidence window is in bins in a range = 0 ... 2000000000 with the bin width of the device (1 ps).

The coincidence counters are not accumulated, i.e. the counter values for the last exposure (see `setExposureTime` above) are returned.

The Counters in the array come in the following channel order with single counts and coincidences:

> 0(5), 1, 2, 3, 4, 1/2, 1/3, 2/3, 1/4, 2/4, 3/4, 1/5, 2/5, 3/5, 4/5, 1/2/3, 1/2/4, 1/3/4, 2/3/4, 1/2/5, 1/3/5, 2/3/5, 1/4/5, 2/4/5, 3/4/5, 1/2/3/4, 1/2/3/5, 1/2/4/5, 1/3/4/5, 2/3/4/5, 1/2/3/4/5

Where e.g. 1/3 is the 7. entry in the array (-> data(6)) for coincidences between channel 1 and 3.

The variable `updates` is the number of data updates by the device since the last call. If you call that function in a loop more often, the counter `updates` will show you if new data is there. See the example `qutag-GetCoincCounter-LivePlotting-example.py` in action.

At last, we print the data and deinitialize the device with

```
# Disconnects a connected device and stops the internal event loop.
qutag.deInitialize()
```

2.3 Get timestamps and check data loss - Example Python / DLL functions

The example `qutag-GetTimestamps-starter_example.py` shows how to retrieve timestamps and be sure, that none are lost.

Let's check if the device lost some data. Timestamps of events detected by the device can get lost if their rate is too high for the USB interface or if the PC is unable to receive the data in time. The quTAG recognizes this situation and signals it to the PC (with high priority). The function checks if a data loss situation is currently detected or if it has been latched since the last call. If you are only interested in the current situation, call the function twice; the first call will delete the latch.

```
dataloss = qutag.getDataLost()
print("dataloss:" + str(dataloss))
```

The next function retrieves the timestamp values of the last n detected events on all channels. All timestamps are from a ring buffer with a variable buffer size.

- The default of the buffer size at the initialization of the python wrapper is 1000000.
- In any other usage of the DLL, use the DLL function TDC_setTimestampBufferSize (default 0 -> 0 data will be returned).
- The bool variable describes if the buffer should be cleared after retrieving the data (True).

```
timestamps = qutag.getLastTimestamps(True)
print("Timestamps in an array:")
print(timestamps)
```

The variable timestamps show our data in an array of:

- an array with all timestamps of the last events in base units 1 ps
- an array with the corresponding channels, range is 0...7 for the channels 1...8
- a variable which shows the number of the valid entries in the above arrays.
  This may be less than the buffer size if the buffer has been cleared.
  !!! If it is the same as the buffer size, the ring buffer is full.
  Probably, the ring buffer was overwritten by new data which was not retrieved yet.

At last, we deinitialize the device with

```
# Disconnects a connected device and stops the internal event loop.
qutag.deInitialize()
```

2.4 Write timestamps to file - Example Python / DLL functions

The example `qutag-WriteTimestamps-starter_example.py` shows how to write timestamps from the quTAG to a file.

First, we import the quTAG wrapper and initialize the device – like in every example.

```
try:
        import QuTAG
except:
        print("Time Tagger wrapper QuTAG.py is not in the search path.")
```

```
# Initialize the quTAG device
qutag = QuTAG.QuTAG()
```

After that, we create a variable `filename` for the name of our output text file.

Then we start writing to the file:

```
# start writing Timestamps from the quTAG
qutag.writeTimestamps(filename,qutag.FILEFORMAT_ASCII)
```

The timestamps come in the base units of 1 ps. All timestamps written are already corrected by the detector delays, see example `qutag-GetHistogramLoop-channelDelay-example.py`.

The channel numbers start with 0 in binary formats, with 1 in ASCII.
A channel number of 100 and higher is associated with the marker input events.
The channel number 104 is a millisecond tick.

The following file formats are available:

- *ASCII*: `FILEFORMAT_ASCII` - Timestamp values (int base units) and channel numbers as decimal values in two comma separated columns. Channel numbers range from 1 to 8 in this format.
- *binary*: `FILEFORMAT_BINARY` - A binary header of 40 bytes, records of 10 bytes, 8 bytes for the timestamp, 2 for the channel number, stored in little endian (Intel) byte order.
- *compressed*: `FILEFORMAT_COMPRESSED` - A binary header of 40 bytes, records of 40 bits (5 bytes), 37 bits for the timestamp, 3 for the channel number, stored in little endian (Intel) byte order. No marker events and timer ticks are stored.
- *raw*: `FILEFORMAT_RAW` - Like binary, but without header. Provided for backward compatibility.

After a short sleep we stop writing the timestamps.

```
# stop writing Timestamps
qutag.writeTimestamps('',qutag.FILEFORMAT_NONE)
```

At last, we print the data and deinitialize the device.

2.5 Create and get histograms - Example Python / DLL functions

The next example **`qutag-GetHistogram-starter_example.py`** shows how to retrieve histograms of the quTAG.

Here we are going to plot the histograms with [`matplotlib`](matplotlib).

First, we import the quTAG wrapper and initialize the device – like in every example.

```
    try:
            import QuTAG
    except:
            print("Time Tagger wrapper QuTAG.py is not in the search path.")
```

```
# Initialize the quTAG device
qutag = QuTAG.QuTAG()
```

After that, we create variables for the input channels we want to create the histogram of.

```
# Choose our start and stop channel
ch_start = 2
ch_stop = 3
```

Now we tell the software to add the desired histogram. In the parameters we name the start channel `ch_start`, the stop channel `ch_stop` and if the histogram should be added `True` or removed `False`.

```
qutag.addHistogram(ch_start,ch_stop,True)
```

After that we give the quTAG some time to accumulate the data with `time.sleep(1).` Therefore, in the first lines of the sample code, we imported `import time`.

Now let's get the histogram data.

```
### Get the histogram of channel ch_start & ch_stop and clear the data with True
rc = qutag.getHistogram(ch_start,ch_stop,True)
```

The function `getHistogram` has three parameters with the start channel `ch_start`, the stop channel `ch_stop` and a boolean variable if we want to clear the histogram data after retrieving it.

The function returns an array with 7 elements. The first element `rc[0]` is also an array with our histogram data with at least binCount elements.

The second variable `r[1]` is an integer with the total number of time differences in the histogram.

`r[2]` and `r[3]` are integers which show the number of time diffs that were smaller than the smallest histogram bin and number of time diffs that were bigger than the biggest histogram bin.

`r[4]` and `r[5]` are integers which show the number of events on the start channel contributing to the histogram and number of events on the stop channel contributing to the histogram.

The last variable `r[6]` shows the total exposure time for the histogram: the time difference between the first and the last event that contribute to the histogram in units of the quTAG time base 1 ps.

In the rest of the example we simply plot the data of the histogram `rc[0]` with `matplotlib` and deinitialize the device with

```
# Disconnects a connected device and stops the internal event loop.
qutag.deInitialize()
```

The example **`qutag-GetHistogramLoop-channelDelay-example.py`** shows how to retrieve histograms in a loop. We also set a delay on one channel every loop, so the histogram is moving in the plot.

2.6 Handle multiple devices - Example Python / DLL functions

The example `qutag-MultipleDevices-starter_example.py` shows how to connect to multiple devices which are plugged in the PC via USB.

> Using multiple devices uses the `tdcmultidev.h`. See the `tdcmultidev.h` file reference in the TDCBase documentation for additional information.

This time we import the quTAG wrapper and use another class for initialization.

```
try:
        import QuTAG
except:
        print("Time Tagger wrapper QuTAG.py is not in the search path.")
```

The following initialization loads another class which starts searching for devices and connects to all of them. The class `QuTAGmulti` is different to the standard class `QuTAG`: in the initialization, the class searches for devices with another DLL function. In the python wrapper it can be called by `QuTAG.discover()` and returns the number of found devices.

> The python function `discover()` is the wrapped DLL function `TDC_discover`. This function initializes the DLL library and searches for connected TDC devices. All existing device connections are closed. The returned parameter is the number of found devices.

and connects to each of them automatically.

> The DLL function `TDC_connect` is called with the device number as parameter. It establishes a connection to the selected device. This is a precondition for all function calls that affect that specific device, in particular for `TDC_addressDevice`. The call implicitly addresses the device. The device number is an integer from `0` - `devCount`, where `devCount` is the return of `TDC_discover`.

```
# Initialize the quTAG devices
qutag = QuTAG.QuTAGmulti()
```

The console log shows us the number of found devices - these does not have to be connected – and shows the addresses of the connected devices.

After that we retrieve the most recent values of the built-in coincidence counters from quTAG from both devices. Therefore, we first address one of the connected devices.

```
qutag.addressDevice(0)
```

The function `addressDevice(0)` addresses the device 0 by the parameter. The device number is given by an increasing integer from 0 up to the number of found of devices, printed before in the initialization process. This is done in the wrapper by calling `QuTAG.discover()`.

Now all the following code addresses the device 0. For instance, we call the count rates, like in chapter 2.2.

```
data,updates = qutag.getCoincCounters()
```

To address the next device, simply call the function `qutag.addressDevice(1)` again with the next parameter 1.

To disconnect one of these devices, call the function `qutag.disconnect(0)` for the device with the address 0.

Be aware, in this chapter the `qutag` is initialized by the class `QuTAGmulti`. At older versions of the wrapper, the class might be not included.

3 Command Line Interface

The command line interface can be used for some simple tasks as a one-time readout of the count rates, writing time stamps to a file or saving histogram data. To use it, open a command prompt, go to the software directory and call tdccli(.exe). Use the parameter -h for additional information.

The CLI program also serves as an example DLL application. The source code is available in userlib\cli.

4 FAQ

4.1 USB Driver Version

Problem	Windows only: The software doesn't find any device. The software claims "Wrong version of USB Driver installed".
Cause	The quTAG is using an USB chip of FTDI. The Software works with FTDI's driver version 1.2. In the meantime, FTDI published version 1.3 which breaks the Software (we are convinced that the version is buggy but were not able to settle that with FTDI). The quTAG software installer is installing 1.2, but Windows sometimes updates to 1.3 autonomously. We don't know when and why that happens but it has sometimes been observed directly after installing the quTAG Software. This applies to Windows 7, 8, and 10.
Solution	The driver version 1.3 has to be removed and replaced by 1.2. To remove, open the device manager, click the device ("FTDI Bridge Device" or "quTAG R3") and disable the driver. Select the option "remove driver from PC" (or similar). Sometimes Windows offers an option "return to old driver version" (or similar) - that also works. Then install again the Software package, the previous installation hasn't to be removed.

4.2 Firmware Version

Problem	The Daisy software warns that the firmware version doesn't match the software version.
Cause	In many cases new software packages come along with new firmware files. The software can work correctly only if the matching firmware is installed.
Solution	Call the flasher tool "nhflash(.exe)" of the new software package and press "Flash" to update the firmware.

4.3 Unexpected Delays

Problem	The time differences between events happening on different input channels are systematically wrong by some nanoseconds.
Cause	quTAGs input signals have slightly different internal runtimes. The "signal delay" function is used to compensate that as well as external runtime differences. On delivery, the delays are not calibrated.
Solution	To remove the runtime difference between two channels, connect them with input signals of known pulse distance and adjust the signal runtime until the quTAG reproduces the known value.

4.4 Low Resolution with Jitter Upgrade

Problem	A quTAG with "Jitter Upgrade" has very low time resolution. Daisy warns about uncalibrated channels.
Cause	The Jitter upgrade requires a calibration procedure that should be repeated from time to time. A quTAG that has never been calibrated exhibits an extremely low time resolution of about 5ns. On delivery, the devices are uncalibrated.
Solution	Perform the calibration procedure according to the handbook for every input channel.

Revision	Date	Changes
1.5.0	2019-11-20	Example for multiple devices added: `qutag-MultipleDevices-starter_example.py`
1.5.0	2019-10-07	Two examples added: `qutag-WriteTimestamps-starter_example.py` & `qutag-GetTimestamps-starter_example.py`
1.5.0	2019-09-27	Description of the Python wrapper