Time Tagger User Manual

Release 1.2.3-local-build

Swabian Instruments

Oct 01, 2020

CONTENTS

CHAPTER

ONE

GETTING STARTED

The following section describes how to get started with your Time Tagger.

First, please install the most recent driver/software which includes a graphical user interface (Web Application) and libraries and examples for C++, Python, .NET, C#, LabVIEW, Matlab and Mathematica.

- Time Tagger software https://www.swabianinstruments.com/time-tagger/downloads/ from our downloads site

You are highly encouraged to read the sections below to get started with the graphical user interface and/or the Time Tagger programming libraries.

In addition, information about the hardware, API, etc. can be found in the menu bar on the left and on our main website: https://www.swabianinstruments.com/time-tagger/.

How to get started with Linux can be found in the *Linux* section.

1.1 Web Application

The Web Application is the provided GUI to show the basic functionality and can be used to do quick measurements.

1. Download and install the most recent Time Tagger software from our downloads site.
2. Start the `Time Tagger Application` from the Windows start menu.
3. The Web Application should show up in your browser.

The Web Application allows you to work with your *Time Tagger* interactively. We will now use the Time Tagger's internal test signal to measure a *cross correlation* between two channels as an example.

1. Click `Add TimeTagger`, click `create` on any of the available Time Taggers
2. Click `Create measurement`, look for `Bidirectional Histogram (Class: Correlation)` and click `Create` next to it.
3. Select `Rising edge 1` for Channel 1 and `Rising edge 2` for Channel 2.
4. Set `binwidth` to 10 ps and leave `n_bins` at 1000, click `initialize`.

The Time Tagger is now acquiring data, but it does not yet have a signal. We will now enable its internal test signal.

1. On the top left, click on the settings wheel next to `Time Tagger`.
2. On the far right, check `Test signal` for channels `1` and `2`, click `Ok`.
3. A Gaussian peak should show up. You can zoom in using the controls on the plot.
4. A Gaussian peak should be displayed. You can zoom in using the controls on the plot.
5. The detection jitter of a single channel is sqrt(2) times the standard deviation of this two-channel measurement (the FWHM of the Gaussian peak is 2.35 times its standard deviation).

You have just verified the time resolution (detection jitter) of your Time Tagger.

Where to go from here...

To learn more about the *Time Tagger* Web Application you are encouraged to consult the following resources.

1. Check out the API documentation in the subsequent chapter.
2. Check out the following sections to get started using the *Time Tagger* software library in the programming language of your choice.
3. Study the code examples in the `[INSTALLDIR]\examples\<language>\` folders of your Time Tagger installation.

1.2 Python

1. Make sure that your *Time Tagger* device is connected to your computer and the *Time Tagger* Web Application is closed.
2. Make sure the *Time Tagger* software and a Python distribution (we recommend **anaconda**) are installed.
3. Open a command shell and navigate to the `.\examples\Python` folder in your *Time Tagger* installation directory
4. Start an **ipython** shell with plotting support by entering `ipython --pylab`
5. Run the **quickstart.py** script by entering `run quickstart`

The script demonstrates a selection of the features provided by the *Time Tagger* programming interface and runs some example measurements using the built-in test signal generator and plots the results.

You are encouraged to open and read the `quickstart.py` file in an editor to understand what it is doing.

The script has many examples which can be followed, including how to:

1. Create an instance called 'tagger' that represents the device.
2. Start the built-in test signal (~0.8 MHz square wave) and apply it to channels 1 and 2
3. Create a time trace of the click rate on channels 1 and 2, let it run for a while and plot the result.
4. Create coarse and fine cross correlation measurements. The coarse measurement shows characteristic peaks at integer multiples of the inverse frequency of the test signal. The fine measurement demonstrates the < 60 ps time resolution.
5. Create virtual channels, use synchronization, the event filter and control the input trigger level.

Now you have learned about the basic functionality of the *Time Tagger* you are encouraged to consult the following resources for more in-depth information.

1. If you have not done so already, have a look at the Python script you just ran.
2. More details about the software interface are covered by the API documentation in the subsequent section

1.3 LabVIEW (via .NET)

A set of examples is provided in `.\examples\LabVIEW\` for LabVIEW 2014 and higher (32 and 64 bit).

1.4 Matlab (wrapper for .NET)

Wrapper classes are provided for Matlab so that native Matlab variables can be used.

The Time Tagger toolbox is automatically installed during the setup. If `Time Tagger` is not available in your Matlab environment try to reinstall the toolbox from `.\driver\Matlab\TimeTaggerMatlab.mltbx`.

The following changes in respect to the .NET library have been made:

- static functions are available through the `TimeTagger` class
- all classes except for the TimeTagger class itself have a `TT` prefix (e.g. `TTCountrate`) to not conflict with any variables/classes in your Matlab environment

An example of how to use the Time Tagger with Matlab can be found in `.\examples\Matlab\`.

1.5 Wolfram Mathematica (via .NET)

Time Tagger functionality is provided to Mathematica via .NET interoperability interface. Please take a look at the examples in `.\examples\Mathematica\`.

1.6 .NET

We provide a .NET class library (32 and 64 bit) for the TimeTagger which can be used to access the TimeTagger from many high-level languages.

The following are important to note:

- Namespace: `SwabianInstruments.TimeTagger`
- the corresponding library `.\driver\xxx\SwabianInstruments.TimeTagger.dll` is registered in the Global Assembly Cache (GAC)
- static functions (e.g. to create an instance of a TimeTagger) are accessible via `SwabianInstruments.TimeTagger.TT`

1.7 C#

A sample project how to use the .NET class library is provided in the `.\examples\Csharp\` folder. Please copy the folder to a folder within the user environment such that files can be written within the folder.

The provided project is a Visual Studio 2017 C# project.

1.8 C++

The provided Visual Studio 2017 C++ project can be found in `.\examples\CXX\`. Using the C++ interface is the most performant way to interact with the TimeTagger as it supports writing custom measurement classes. But it is more elaborate compared to the other high-level languages. Please visit `.\documentation\Time Tagger C++ API Manual.pdf` for more details on the C++ API.

Note:

- the C++ headers are stored in the `.\driver\include\` folder
- the final assembly must link `.\driver\xxx\TimeTagger.lib`
- the library `.\driver\xxx\TimeTagger.dll` is linked with the shared v141 Visual Studio runtime (`/MD`)

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

2.1 Requirements

2.1.1 Operating System

Windows Windows 7 or higher

We provide separate Windows installers for 32 and 64 bit systems.

2.1.2 Installation

Download and install the most recent Time Tagger software from our downloads site.

Connect the *Time Tagger* to your computer with the USB cable.

You should now be ready to use your *Time Tagger*.

2.1.3 Web Application

The Web Application is the provided GUI to show the basic functionality and can be used to do quick measurements. See *Getting Started: Web application* for further information.

2.1.4 Programming Examples

The Time Tagger installer provides programming examples for Python, Matlab, Mathematica, LabVIEW, C#, and C++ within the `.\examples\<language>\` folders of your Time Tagger installation. See *Getting Started: Examples* for further information.

CHAPTER

THREE

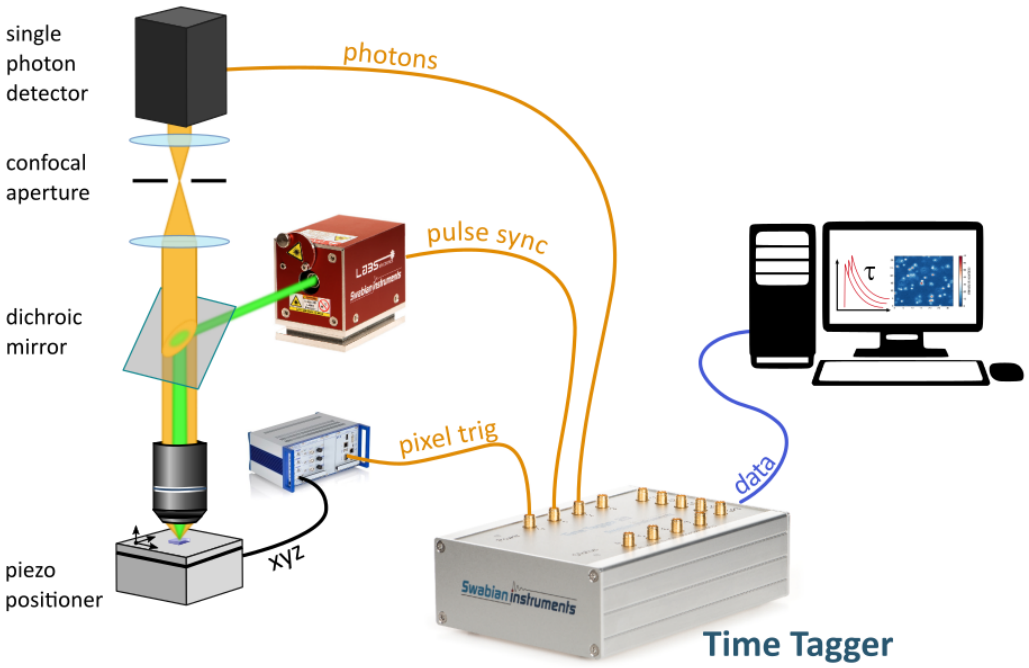
TUTORIALS

3.1 Confocal Fluorescence Microscope

This tutorial guides you through setting up a data acquisition for a typical confocal microscope controlled with Swabian Instruments' Time Tagger. In this tutorial, we will use Time Tagger's programming interface to define the data acquisition part of a scanning microscope. We will make no specific assumption of how the position scanning system is implemented except that it has to provide suitable signals detailed in the text.

The basic principle of confocal microscopy is that the light, collected from a sample, is spatially filtered by a confocal aperture, and only photons from a single spot of a sample can reach the detector. Compared to conventional microscopy, confocal microscopy offers several advantages, such as increased image contrast and better depth resolution, because the pinhole eliminates all out-of-focus photons, including stray light.

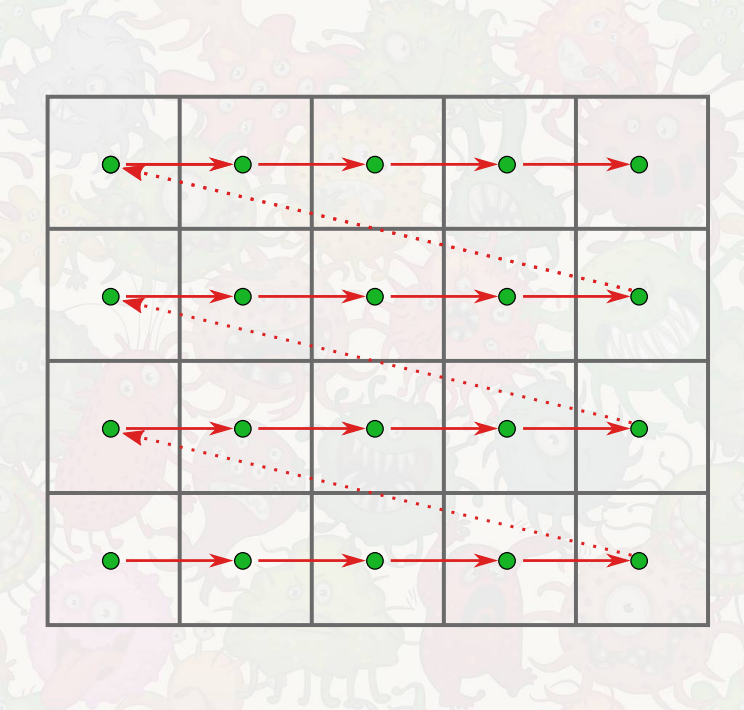
The following drawing shows a typical confocal fluorescence microscope setup.



In this setup, the objective focuses the excitation light from the laser at the fluorescent sample and, at the same time, collects the resulting emission. The emission photons pass through the confocal aperture and arrive at the single-photon detector (SPD). For every detected photon, the SPD produces a voltage pulse at its output, namely a photon pulse.

Image from a raster scan

In the confocal microscopy, the detection area is a small diffraction-limited spot. Therefore, to record an image, one has to scan the sample surface point-by-point and record the detector signal at every location. The majority of scanning microscopes employ a raster scan path that visits every point on sample step-by-step and line-by-line. The figure below visualizes the travel path in a typical raster scan.



In the figure above, the scan starts from the bottom-left corner and proceeds horizontally in steps. At each scan position, the scanner has to wait for arbitrary integration time to allow sufficient photon collection. This process stops when the scanner reaches the top-right point.

Along the scan path, the positioner generates a pulse for every new sample position. In the following text, we will call this signal a pixel pulse.

To measure a confocal fluorescence image, the arrival times of the following three signals must be recorded: photon pulses, laser pulses, and pixel pulses.

3.1.1 Time Tagger configuration

The Time Tagger library includes several measurement classes designed for confocal microscopy.

We will start by defining channel numbers and store them in variables for convenience.

```
PIXEL_START_CH = 1 # Rising edge on input 1
PIXEL_END_CH = -1  # Falling edge on input 1
LASER_CH = 2
SPD_CH = 3
```

Now let's connect to the Time Tagger.

```
tt = createTimeTagger()
```

The Time Tagger hardware allows you to specify a trigger level voltage for each input channel. This trigger level, always applies for both, raising and falling edges of an input pulse. Whenever the signal level crosses this trigger level, the Time Tagger detects this as an event and stores the timestamp. It is convenient to set the trigger level to half a signal amplitude. For example, if your laser sync output provides pulses of 0.2 Volt amplitude, we set the trigger level to 0.1 V on this channel. The default trigger level is 0.5 Volt.

```
tt.setTriggerLevel(PIXEL_START_CH, 0.5)
tt.setTriggerLevel(LASER_CH, 0.1)
```

The Time Tagger allows for delay compensation at each channel. Such delays are inevitably present in every measurement setup due to different cable lengths or inherent delays in the detectors and laser sync signals. It is worth noting that a typical coaxial cable has a signal propagation delay of about 5 ns/m.

Let's suppose that we have to delay the laser pulse by 6.3 ns, if we want to align it close to the arrival time of the fluorescence photon pulse. Using the Time Tagger's API, this will look like:

```
tt.setInputDelay(LASER_CH, 6300)  # Delay is always specified in picoseconds
tt.setInputDelay(SPD_CH, 0)       # Default value is: 0
```

Now we are finished with setting up the Time Tagger hardware and are ready to proceed with defining the measurements.

3.1.2 Intensity scanning microscope

In this section, we start from an easy example of only counting the number of photons per pixel and spend some time on understanding how to use the pixel trigger signal. The Time Tagger library contains the generic *CountBetweenMarkers* measurement that has all the necessary functionality to implement the data acquisition for a scanning microscope.

For the *CountBetweenMarkers* measurement, you have to specify on which channels the photon and the pixel pulses arrive. Also, we have to specify the total number of points in the scan, which is the number of pixels in the final image. Furthermore, we assume that the pixel pulse edges indicate when to start, and when to stop counting photons and the pulse duration defines the integration time. If your scanning system generates pixel pulses of a different format, take a look at the section *Alternative pixel trigger formats*.

As a first step, we create a measurement object with all the necessary parameters provided.

```
nx_pix = 300
ny_pix = 200
n_pixels = nx_pix * ny_pix

cbm = CountBetweenMarkers(tt, SPD_CH, PIXEL_START_CH, PIXEL_STOP_CH, n_pixels)
```

The measurement is now prepared and waiting for the signals to arrive. The next step is to send a command to the piezo-positioner to start scanning and producing the pixel pulses for each location.

```
scanner.scan(
    x0=0, dx=1e-5, nx=nx_pix,
    y0=0, dy=1e-5, ny=ny_pix,
)
```

Note: The code above introduces a *scanner* object which is not part of the Time Tagger library. It is an example of a hypothetical programming interface for a piezo-scanner. Here, we also assume that this call is non-blocking, and the

script can continue immediately after starting the scan.

After we started the scanner, the Time Tagger receives the pixel pulses, counts the events at each pixel, and stores the count in its internal buffer. One can read the buffer content periodically without disturbing the acquisition, even before the measurement is completed. Therefore, you can see the intermediate results and visualize the scan progress.

The resulting data from the *CountBetweenMarkers* measurement is a vector. We have to reorganize the elements of this vector according to the scan path if we want to display it as an image. For the raster scan, this reorganization can be done by a simple reshaping of the vector into a 2D array.

The following code gives you an example of how you can visualize the scan process.

```
while scanner.isScanning():
    counts = cbm.getData()
    img = np.reshape(counts, nx_pix, ny_pix)
    plt.imshow(img)
    plt.pause(0.5)
```

3.1.3 Fluorescence Lifetime Microscope

In the section *Intensity scanning microscope*, we completely discarded the time of arrival for photon and laser pulses. The Time Tagger allows you to record a fluorescence decay histogram for every pixel of the confocal image by taking into account the time difference between the arrival of the photon and laser pulses. This task can be achieved using the *FLIM* or *TimeDifferences* measurements from the Time Tagger library. In this subsection, we will use the *FLIM* measurement.

The *FLIM* measurement calculates the time differences between laser and photon pulses and accumulates them in a histogram for every pixel. The measurement class constructor requires imaging and timing parameters, as shown in the following code snippet.

```
nx_pix = 300   # Number of pixels along x-axis
ny_pix = 200   # Number of pixels along y-axis
binwidth = 50  # in picoseconds
n_bins = 2000  # number of bins in a histogram
n_pixels = nx_pix * ny_pix # number of histograms

flim = Flim(tt, SPD_CH, LASER_CH, PIXEL_START_CH, binwidth, n_bins, n_pixels)
```

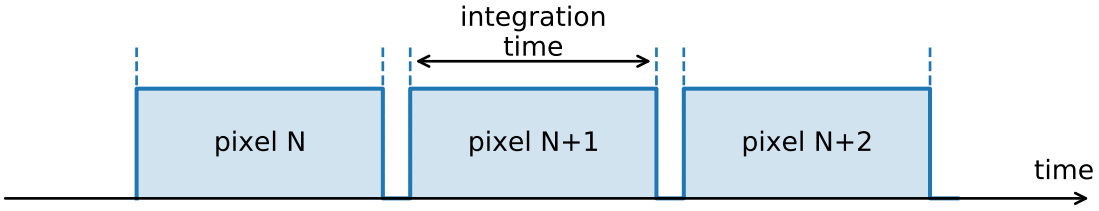
Now we start the scanner and wait until the scan is completed. During the scan, we can read the current data and display it in real time.

```
while scanner.isScanning():
    counts = flim.getData()
    img3D = np.reshape(counts, n_bins, nx_pix, ny_pix) # Fluorescence image cube

    # User defined function that estimates fluorescence lifetime for every pixel
    flimg = get_lifetime(img3D)

    plt.imshow(flimg)
    plt.pause(0.5)
```

3.1.4 Alternative pixel trigger formats

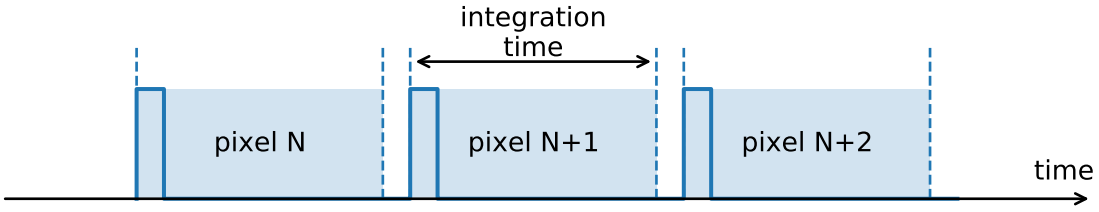
What if a piezo scanner provides a different trigger signal compared to considered in the previous sections? In this section, we look into a few common types of trigger signals and how to adapt our data acquisition to make them work.

Pixel pulse width defines the integration time



The case when the pulse width defines the integration time has been considered in the previous subsections.

Pixel pulse indicates the pixel start



When a pixel pulse has a duration different from the desired integration time, we must define the integration time manually. One way would be to record all events until the next pixel pulse and rely on a strictly fixed pixel pulse period. Alternatively, we can create a well-defined time window after each pixel pulse, so the measurement system becomes insensitive to the variation of the pixel pulse period.

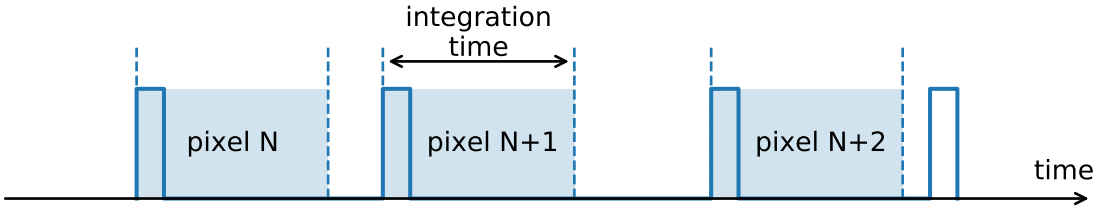
One can define the time window using the *DelayedChannel* which provides a delayed copy of the leading edge for the pixel pulse.

```
integr_time = int(1e10)  # Integration time of 10 ms in picoseconds
delayed_vch = DelayedChannel(tt, PIXEL_START_CH, integr_time)
PIXEL_END_CH = delayed_vch.getChannel()

cbm = CountBetweenMarkers(tt, SPD_CH, PIXEL_CH, PIXEL_END_CH, n_pixels)
```

The approach with using *DelayedChannel* allows for a constant integration time per pixel even if the pixel pulses do not occur at a fixed period. For instance, in a raster scan, more time is required to move to the beginning of the next line (fly-back time) compared to the pixel time.

Warning: You have to make sure that pixel pulses do not appear before the end of the integration time for the previous pixel.

FLIM with non-periodic pixel trigger



In some cases, a scanner generates the pixel pulses with no strictly defined period. However, most scanning measurements require constant integration time for every pixel. Compared to *CountBetweenMarkers*, the *Flim* and *TimeDifferences* measurements do not have a *PIXEL_END* marker and accumulate the histogram for every pixel until the next pixel pulse is received. If this behavior is undesired, or if your pixel pulses are not periodic, you will need to gate your detector to guarantee a constant integration time.

The Time Tagger library provides you with the necessary tools to enforce a fixed integration time when using the *Flim* measurement. Gating the detector events can be done with the *GatedChannel*. The example code is provided below.

```
integr_time = int(1e10)  # Integration time of 10 ms in picoseconds
delayed_vch = DelayedChannel(tt, PIXEL_START_CH, integr_time)
PIXEL_END_CH = delayed_vch.getChannel()

gated_vch = GatedChannel(tt, SPD_CH, PIXEL_START_CH, PIXEL_END_CH)
GATED_SPD_CH = gated_vch.getChannel()

flim = Flim(tt, GATED_SPD_CH, LASER_CH, PIXEL_START_CH, binwidth, n_bins, n_pixels)
```

CHAPTER

FOUR

SYNCHRONIZER

4.1 Overview

The Swabian Instruments' Synchronizer allows for connecting up to 8 Time Taggers to expand the number of available channels. The Synchronizer generates a clock and synchronization signal to establish a common time-base on all connected Time Taggers. The Time Tagger software engine creates a layer of abstraction: the synchronized Time Taggers appear as one device with a combined number of input channels.

4.2 Requirements

Successful synchronization of your Time Taggers requires:

- You have obtained the Synchronizer hardware.
- Your Time Tagger Ultra has hardware version 1.2 or higher. In case you have an older device and want to synchronize it with more units, please contact our support or sales team www.swabianinstruments.com/contact .
- Your PC has a sufficient number of USB3 ports for direct connection of every Time Tagger. The Synchronizer itself does not require a USB connection.
- You have a sufficient number of SMA cables of the same length. You need three cables for each Time Tagger. For more details, see in the section *Cable connections*.
- You have installed the Time Tagger software version 2.6.6 or newer.

4.3 Cable connections

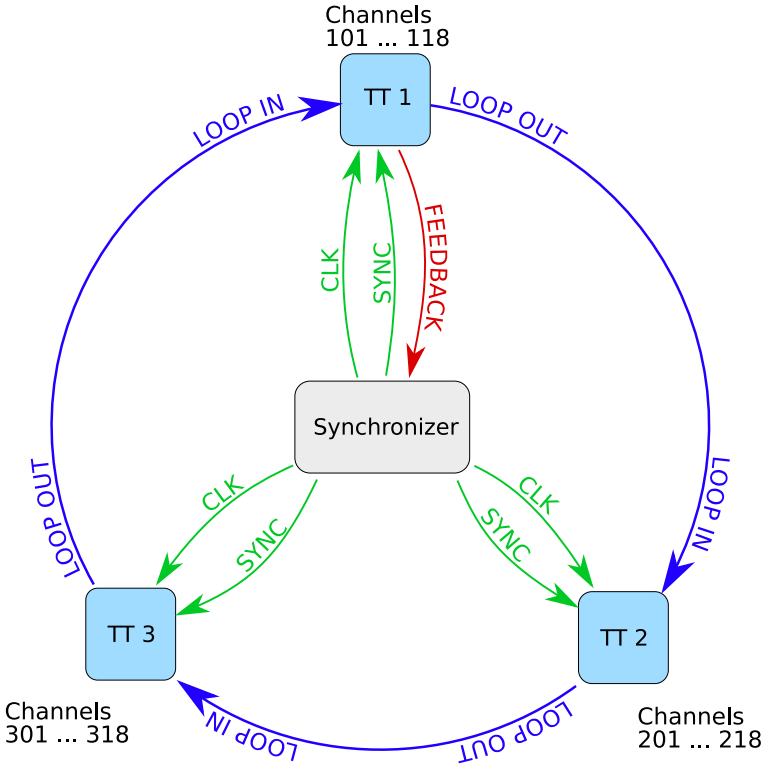
The Synchronizer provides a common clock signal for every Time Tagger as well as the synchronization signals. Furthermore, Time Taggers have to be connected to each other in a loop. The connection sequence in the loop defines the channel numbering order. An additional feedback signal is required to identify which of the Time Taggers in the loop is the first.

Note: After the release of the Synchronizer, we have changed the connector labels on the front panel of Time Tagger Ultra. In this section, we use the new labeling scheme, while showing the corresponding old labels in brackets: *NEW_LABEL (OLD_LABEL)*.

Table 1: Connections between the Synchronizer and Time Taggers

Synchronizer	Time Tagger	Description
CLK OUT <N>	*CLK IN (CLK)*	500 MHz clock
SYNC OUT <N>	*SYNC IN (AUX IN 1)*	Synchronization data
FDBK IN	*FDBK OUT (AUX OUT 2)*	Feedback from one Time Tagger

Every Time Tagger should have its *LOOP OUT (AUX OUT 1)* connected to the *LOOP IN (AUX IN 2)* of next Time Tagger, eventually forming a signal loop. The following diagram visualizes the connections required for the synchronization of three Time Taggers.



Warning: For reliable synchronization, the cables for *CLK* and *SYNC* signals shall have a length difference below 4 cm. We recommend using the same cable type for these two signals.

Additionally, we recommend connecting every Time Tagger directly to a USB3 port on the same computer. If your computer does not have a sufficient number of USB3 ports, avoid using USB hubs as they limit the data bandwidth available for every Time Tagger. Instead, please install an additional USB controller card into your computer. While there is a wide variety of USB3 controllers, you have to look for one that can deliver full USB3 bandwidth at every USB port simultaneously. Typically, such USB controllers have an individual chip for each USB port and require a PCIe x4 slot on the computer's motherboard..

4.3.1 Using an external reference clock

The Synchronizer has a built-in high accuracy and low noise reference oscillator and distributes the clock signals to all attached Time Taggers. In case you want to use your external reference clock, you have to connect it to the *REF IN* connector of the Synchronizer. Additionally, the Synchronizer can supply 10 MHz reference signal through its *REF OUT* output. Note that *REF OUT* is disabled when an external reference signal is present at the *REF IN*.

Table 2: Requirements to the reference signal at *REF IN*.

Parameter	Value
Coupling	AC
Amplitude	0.3 . . . 5.0 Vpp
Frequency	10 MHz
Impedance	50 Ohm

Table 3: Signal parameters at *REF OUT*.

Parameter	Value
Coupling	AC
Amplitude	3.3 Vpp (1 Vpp @ 50 Ohm)
Frequency	10 MHz

4.4 Software and channel numbering

The Time Tagger software engine automatically recognizes if a Time Tagger belongs to a synchronized group. It will also automatically open a connection to all other Time Taggers in the group and present all devices as a single Time Tagger. There is no specific "master" device, and the connection to the synchronized group can be initiated from any of the member Time Taggers.

The connection is opened as usual using *createTimeTagger()*, and optionally you can specify the serial number of the Time Tagger.

```
tagger = createTimeTagger()
```

The *tagger* object provides a common interface for the whole synchronization loop, and all programming is done in the same way as for a single Time Tagger. Note that, compared to a single Time Tagger, the channel numbering scheme is modified for easy identification by a user. The channel number consists of the Time Tagger number in the loop and the input number on the front panel. The channel number formula is

```
CHANEL_NUMBER = TT_NUMBER*100 + INPUT_NUMBER
```

As an example, let us assume we have three Time Tagger Ultra 18 in a synchronization loop. The Time Tagger that provides the feedback signal to the Synchronizer has sequence number 1, and its channel numbers will be from 101 to 118. The channels of the next Time Tagger will have numbers from 201 to 218, and so forth.

Note: In case the channel numbers on your Time Tagger Ultra start with 0, in the synchronized group, the channel 0 will appear as N01, where N is the Time Tagger number. See more about channel numbering scheme in the section *Channel Number Schema 0 and 1*.

You can request the complete list of available channels with the *TimeTagger.getChannelList()* method.

```
from TimeTagger import createTimeTagger, TT_CHANNEL_RISING_EDGES

# Connect to any of the synchronized Time Taggers
tagger = createTimeTagger()

# Request a list of all positive edge channels
chan_list = tagger.getChannelList(TT_CHANNEL_RISING_EDGES)
print(chan_list)
>> [101, 102, ... , 317, 318]
```

4.4.1 Incomplete cable connections

The software engine attempts to detect incorrect or incomplete connections of the cables in the synchronization loop. In case some connections are missing or were disconnected during operation, the software engine will show a warning and the data transmission from the disconnected Time Tagger will be filtered out until a valid connection is restored. Issues with the cable connections and synchronization status are indicated using the status LEDs on the front panel of the Synchronizer and the Time Tagger. See more in section *Status LEDs and troubleshooting*.

4.4.2 Buffer overflows

The synchronization loop also propagates the buffer overflow state from any Time Tagger to all members of the loop. On the software side, the buffer overflow has the same effect as for a single Time Tagger. See, *Overflows*.

4.5 Limitations

4.5.1 Conditional filter

The conditional filter cannot be applied across synchronized devices. However, it can still be enabled for each Time Tagger independently.

In case you want to use the conditional filter across devices, you have to send the signal to be filtered (for example, your laser sync) to every Time Tagger where trigger signals are connected. In software, you have to choose the corresponding input for time difference measurements.

4.5.2 Internal test signal

The internal test-signal generator is a free-running oscillator independent from the system clock. Therefore, the test signals are not correlated between different Time Taggers, even if the synchronization loop is set up correctly. If you try to measure a correlation with the internal test signal across two different Time Taggers, you will see a flat histogram. On the other hand, performing the same measurement with two input channels of the same Time Tagger will result in a jitter-limited correlation peak.

4.6 Status LEDs and troubleshooting

The front panel of the Synchronizer has several LEDs that indicate operation status.

LED	Color	Description
Power	dark	No power provided
–	solid green	Powered on
Status	dark	Warming up
–	solid green	Normal operation.
FDBK IN	solid green	Normal operation
–	solid red	Invalid feedback signal
REF IN	dark	No external reference signal
–	solid green	Valid 10 MHz reference signal
–	solid red	Invalid reference signal
REF OUT	dark	Output is disabled when using external reference signal
–	solid green	Output enabled

The LEDs of the Time Tagger Ultra also indicate the state of the synchronization loop. See more details in section *Status LEDs*.

CHAPTER

FIVE

HARDWARE

5.1 Input channels

The *Time Tagger* has 8 or 18 input channels (SMA-connectors). The electrical characteristics are tabulated below. Both rising and falling edges are detected on the input channels. In the software, rising edges correspond to channel numbers 1 to 8 (Ultra: 1 to 18) and falling edges correspond to respective channel numbers -1 to -8 (Ultra: -1 to -18). Thereby, you can treat rising and falling edges in a fully equivalent fashion.

5.1.1 Electrical characteristics

Property	Time Tagger 20	Time Tagger Ultra
Termination	50 Ohm	50 Ohm
Input voltage range	0.0 to 5.0 V	-5.0 to 5.0 V
Trigger level range	0.0 to 2.5 V	-2.5 to 2.5 V
Minimum signal level	100 mV	100 mV
Minimum pulse width	1.0 ns	0.5 ns

5.2 Data connection

The *Time Tagger 20* is powered via the USB connection. Therefore, you should ensure that the USB port is capable of providing the full specified current (500 mA). A USB >= 2.0 data connection is required for the performance specified here. Operating the device via a USB hub is strongly discouraged. The *Time Tagger 20* can stream about 8 M tags per second.

The data connection of the *Time Tagger Ultra* is USB 3.0. Therefore the number of tags steamed to the PC can exceed 65 M tags per second. The actual number highly depends on the performance of the CPU the *Time Tagger Ultra* is connected to and the evaluation methods involved.

5.3 Status LEDs

The *Time Tagger* has two LEDs showing status information. A green LED turns on when the USB power is connected. An RGB LED shows the information tabulated below.

green	firmware loaded
blinking green-orange	time tags are streaming
red flash (0.1 s)	an overflow occurred
continuous red	repeated overflows

Table 1: LED next to the *CLK* input

Color	Description
dark	No clock signal
solid green	Valid reference or synchronization signal
solid red	Invalid reference frequency
blinking red	Invalid signal at SYNC IN (AUX IN 1)
blinking yellow	Invalid signal at LOOP IN (AUX IN 2)

5.4 Test signal

The *Time Tagger* has a built-in test signal generator that generates a square wave with a frequency in the range 0.8 to 1.0 MHz. You can apply the test signal to any input channel instead of an external input, this is especially useful for testing, calibrating and setting up the *Time Tagger* initially.

5.5 Virtual channels

The architecture allows you to create virtual channels, e.g., you can create a new channel that represents the sum of two channels (logical OR), or coincidence clicks of two channels (logical AND).

5.6 Synthetic input delay

You can introduce an input delay for each channel independently. This is useful if the relative timing between two channels is important e.g., to compensate for propagation delay in cables of unequal length. The input delay can be set individually for rising and for falling edges.

5.7 Synthetic dead time

You can introduce a synthetic dead time for each channel independently. This is useful when you want to suppress consecutive clicks that are closely separated, e.g., to suppress after-pulsing of avalanche photodiodes or to suppress too high data rates. The dead time can be set individually for rising and for falling edges.

5.8 Conditional Filter

The Conditional Filter allows you to decrease the time tag rate without losing those time tags that are relevant to your application, for instance, where you have a high-frequency signal applied to at least one channel. Examples include fluorescence lifetime measurements or optical quantum information and cryptography, where you want to capture synchronization clicks from a high repetition rate excitation laser.

To reduce the data rate, you discard all synchronization clicks, except those that follow after one of your low rate detector clicks, thereby forming a reduced time tag stream. The software processes the reduced time tag stream in the exact same fashion as the full time tag stream.

This feature is enabled by the Conditional Filter. As all channels on your Time Tagger are fully equivalent, you can specify which channels are filtered and which channels are used as triggers that enable the transmission of a subsequent tag on the filtered channels.

The time resolution of the filter is the very same as the dead time of the channels (Time Tagger 20: 6 ns, Time Tagger Ultra: 2 ns).

To ensure deterministic filter logic, the physical time difference between the filtered channels and triggered channels must be larger than +/- (deadtime + 3 ns). The Conditional Filter also works in the regime when signals arrive almost simultaneously, but one has to be aware of a few details described below. Note also that software-defined input delays as set by the method `setInputDelay()` do not apply to the Conditional Filter logic.

More details and explanations can be found in the In-Depth Guide More details can be found in the *In Depth Guide: Conditional Filter*.

5.9 Bin equilibration

The discretization of electrical signals is never perfect. In time-to-digital conversion, this manifests as small differences (few ps) of the bin sizes inside the converter that even varies from chip to chip. This imperfection is inherent to any time-to-digital conversion hardware. It is usually not apparent to the user. However, when correlations between two channels are measured on short time scales you might see this as a weak periodic ripple on top of your signal. We reduce the effect of this in the software at the cost of a decrease of the time resolution by $\sqrt{2}$. This feature is enabled by default. If your application requires time resolution down to the jitter limit, you can disable this feature.

5.10 Overflows

The *Time Tagger 20* is capable of continuous streaming of about 8 million tags per second on average. For the *Time Tagger Ultra* continuous tags streamed can exceed 65 million tags per second depending on the CPU the Time Tagger is attached to and the evaluation methods involved. Higher data rates for short times will be buffered internally so that no overflow occurs. This internal buffer is limited, therefore, if continuous higher data rates arise, data loss occurs and parts of the time tags are lost. The hardware allows you to check with `timeTagger.getOverflows()` whether an overflow condition has occurred. If no overflow is returned, you can be confident that every time tag is received.

Note: When overflows occur, Time Tagger will still produce valid blocks of data and discard the invalid tags in between. Your measurement data may still be valid, albeit, your acquisition time will likely increase.

5.11 External Clock Input

The external clock input can be used to synchronize different devices. The input clock frequency must be 1/6 GHz (approx. 167 MHz) for the Time Tagger 20 and 10 MHz for the Time Tagger Ultra.

As soon as this frequency is applied to the EXT CLK input, the Time Taggers are locked to it. The lock status can be read off the LED color:

- Time Tagger 20
  - status led stays or blinks white when not in overflow mode (red)
- Time Tagger Ultra
  - CLK led green: locked, red: wrong frequency

CLK Input requirements:

- Time Tagger 20
  - Hardware Version <= 2.1
    - 0 to 5V into 50 Ohm, 0 to 2 V recommended
  - Hardware Version >= 2.2
    - 100 mVpp up to 3 Vpp AC coupled into 50 Ohm, 500 mVpp recommended
- Time Tagger Ultra
  - 100 mVpp - 3 Vpp AC coupled into 50 Ohm, 500 mVpp recommended

Performance:

The input clock signal must have a very low jitter to provide the specified performance of the Time Tagger. Please note that the timing specifications for the Time Tagger Ultra with respect to other devices on the same clock are only met from hardware version 2.3 on.

Caution: In order to reach the specified input jitter for the Time Tagger with an external clock, the input signals must be uncorrelated to the external clock.

5.12 Synchronization signals - Time Tagger Ultra only

Up to 8 Time Tagger Ultra units can be synchronized in such a way that they behave like a unified Time Tagger. This requires additional hardware, the Swabian Synchronizer. The Synchronizer uses the additional hardware connections: SYNC IN, LOOP IN, LOOP OUT and FDBK OUT (see *Synchronizer*).

Warning: On Time Tagger Ultra units sold before September 2020, the synchronization signals use the ports labeled AUX IN 1, AUX IN 2, AUX OUT 1, AUX OUT 2. A mapping of the signal names is included in the Synchronizer documentation (see *Synchronizer*). If you own one of these units and would like to have a sticker to update your labels, please reach out to the Swabian Instruments support .

5.13 General purpose IO (GPIO) - Time Tagger Ultra only

Starting from the Time Tagger v2.6.6, the general purpose inputs and outputs on Time Tagger Ultra are used for synchronization signals. New Time Tagger Ultra devices will have an updated labeling of these IO ports. See, *Synchronizer*

5.14 General purpose IO (GPIO) - Time Tagger 20 only

The Time Tagger 20 is equipped with four general purpose io ports that interface directly with the system's FPGA. These are reserved for future implementations.

CHAPTER

SIX

SOFTWARE OVERVIEW

The heart of the *Time Tagger* software is a multi-threaded driver that receives the time tag stream and feeds it to all running measurements. Measurements are small threads that analyze the time tag stream each in their own way. For example, a count rate measurement will extract all time tags of a specific channel and calculate the average number of tags received per second; a cross-correlation measurement will compute the cross-correlation between two channels, typically by sorting the time tags in histograms, and so on. This is a powerful architecture that allows you to perform any thinkable digital time domain measurement in real time. You have several choices on how to use this architecture.

6.1 Web application

The easiest way of using the *Time Tagger* is via a web application that allows you to interact with the hardware from a web browser on your computer or a tablet. You can create measurements, get live plots, and save and load the acquired data from within a web browser.

6.2 Precompiled libraries and high-level language bindings

We have implemented a set of typical measurements including count rates, auto correlation, cross correlation, fluorescence lifetime imaging (FLIM), etc.. For most users, these measurements will cover all needs. These measurements are included in the C++ API and provided as precompiled library files. To make using the Time Tagger even easier, we have equipped these libraries with bindings to higher-level languages (Python, Matlab, LabVIEW, .NET) so that you can directly use the Time Tagger from these languages. With these APIs you can easily start a complex measurement from a higher-level language with only two lines of code. To use one of these APIs, you have to write the code in the high-level language of your choice. Refer to the chapters *Getting Started* and *Application Programmer's Interface* if you plan to use the Time Tagger in this way.

6.3 C++ API

The underlying software architecture is provided by a C++ API that implements two classes: one class that represents the Time Tagger and one class that represents a base measurement. On top of that, the C++ API also provides all predefined measurements that are made available by the web application and high-level language bindings. To use this API, you have to write and compile a C++ program.

CHAPTER

SEVEN

APPLICATION PROGRAMMER'S INTERFACE

7.1 Overview

The Time Tagger API provides methods to control the hardware and to create *measurements* that are hooked onto the time tag stream. It is written in C++ and we also provide wrapper classes for several common higher-level languages (Python, Matlab, LabVIEW, .NET). Maintaining this transparent equivalence between different languages simplifies documentation and allows you to choose the most suitable language for your experiment. The API includes a set of standard *measurements* that cover common tasks relevant to photon counting and time-resolved event measurements. These classes will most likely cover your needs and, of course, the API provides you a possibility to implement your own custom measurements. Custom measurements can be created in one of the following ways:

- Subclassing the `IteratorBase` class (best performance, but only available in the C++ and Python API - see example in the installation folder)
- Using the *`TimeTagStream`* measurement and processing the raw time tag stream.
- Offline processing when you store timetags into a file using *`FileWriter`* and then read the resulting file to perform desired analysis of the timetags. This also enables to keep a record of the complete chronology of the events in your experiment.

7.1.1 Examples

Often the fastest way to get an impression on the API is through the examples.

Measuring cross-correlation

The code below shows a simple but operational example on how to perform a cross-correlation measurement with the Time Tagger API. In fact, such simple code is already sufficient to perform real-world experiments in a lab.

```
# Create an instance of the TimeTagger
tagger = createTimeTagger()

# Adjust trigger level on channel 2 to 0.25 Volt
tagger.setTriggerLevel(2, 0.25)

# Add time delay of 123 picoseconds on the channel 3
tagger.setInputDelay(3, 123)

# Create Correlation measurement for events in channels 2 and 3
corr = Correlation(tagger, 2, 3, binwidth=10, n_bins=1000)
```

(continues on next page)

(continued from previous page)

```
# Wait for some time to accumulate the data
pause(1)

# Read the correlation data
data = corr.getData()
```

Using virtual channels

Time Tagger API implements on-the-fly timetag processing through *virtual channels*. The following example shows how timetags from two different real channels can be combined into one virtual channel.

```
tagger = createTimeTagger()

# Enable internal generator to channels 1 and 2. Frequency ~800 kHz.
tagger.setTestSignal([1,2], True)

# Create virtual channel that combines timetags from real inputs 1 and 2
vc = Combiner(tagger, [1, 2])

# Create countrate measurement at channels 1, 2 and the "combiner" channel
rate = Countrate(tagger, [1, 2, vc.getChannel()])

# Wait and print the countrate all three channels
pause(1)
print(rate.getData())

>> [ 800008.81  800008.81 1600017.62]
```

From the results, we see that the combined event rate is a sum of the event rates at both input channels, as expected.

Using multiple Time Taggers

You can use multiple Time Taggers on one computer simultaneously. In this case, you usually want to associate your instance of the *TimeTagger* class to the Time Tagger device. This is done by specifying the serial number of the device, an optional parameter, to the factory function *createTimeTagger()*.

```
tagger_1 = createTimeTagger("123456789ABC")
tagger_2 = createTimeTagger("123456789XYZ")
```

The serial number of a physical Time Tagger is a string of digits and letters (every Time Tagger has a unique hardware serial number). It is printed on the label at the bottom of the Time Tagger hardware. In addition, the *scanTimeTagger()* method shows the serial numbers of the connected but not instantiated Time Taggers. It is also possible to read the serial number for a connected device using *TimeTagger.getSerial()* method.

You can find more examples supplied with the TimeTagger software. Please see the `examples\<language>` subfolder of your *Time Tagger* installation. Usually, the installation folder is `C:\Program Files\Swabian Instruments\Time Tagger`.

7.1.2 Units of measurement

Time is measured and specified in picoseconds. Timetags indicate time since device start-up which is represented by a 64-bit integer number. Note that this implies that the time variable will rollover once approximately every 107 days. This will most likely not be relevant to you unless you plan to run your software continuously over several months and you are taking data at the instance when the rollover is happening.

Analog voltage levels are specified in Volts.

7.1.3 Channel numbers

You can use the Time Tagger to detect both rising and falling edges. Throughout the software API, the rising edges are represented by positive channel numbers starting from 1 and the falling edges are represented by negative channel numbers. Virtual channels will automatically obtain numbers higher than the positive channel numbers.

The Time Taggers delivered before mid 2018 have a different channel numbering. More details can be found in the *Channel Number Schema 0 and 1* section.

7.1.4 Unused channels

There might be the need to leave a parameter undefined when calling a class constructor. Depending on the programming language you are using, you pass an undefined channel via the static constant *CHANNEL_UNUSED* which can be found in the `TT` class for .NET and in the `TimeTagger` class in Matlab.

7.2 Module constants

CHANNEL_UNUSED
Can be used instead of a channel number when no specific channel is assumed. In MATLAB use `TimeTagger.CHANNEL_UNUSED`.

7.3 Module functions

createTimeTagger([*serial=''*])
Establishes the connection to a first available Time Tagger device and creates a *TimeTagger* object. Optionally, the connection to a specific device can be achieved by specifying the device serial number.

In MATLAB the *TimeTagger* object is created by instantiating the class directly as `tagger = TimeTagger([serial])`.

Parameters **serial** (*string*) – Serial number string of the device or empty string

Returns TimeTagger object

Return type *TimeTagger*

createTimeTaggerVirtual()
Creates a virtual Time Tagger object. Virtual Time Tagger uses time-tag dump file(s) as a data source instead of Time tagger hardware. This allows you to use all Time Tagger library measurements for offline processing of the dumped time tag stream. For example, you can repeat the analysis of your experiment with different parameters, like binwidths,

In MATLAB the *TimeTaggerVirtual* object is created by instantiating the class directly as `tagger = TimeTaggerVirtual()`.

Returns TimeTaggerVirtual object

Return type *TimeTagger*

freeTimeTagger(*tagger*)
Releases all Time Tagger resources and terminates the active connection.

Parameters **tagger** (TimeTagger) – TimeTagger object to disconnect

freeAllTimeTagger()
Releases all Time Tagger resources and terminates the active connection of all Time Taggers.

scanTimeTagger()
Returns a list of the serial numbers of the connected but not instantiated Time Taggers.

In MATLAB this function is accessible as `TimeTagger.scanTimeTagger()`.

Returns List of serial numbers

Return type list(string)

setLogger(*callback*)
Registers a callback function, e.g. for customized error handling. Please see the examples in the installation folder on how to use it.

setTimeTaggerChannelNumberScheme(*int scheme*)
Selects whether the first physical channel starts with 0 or 1

`TT_CHANNEL_NUMBER_SCHEME_AUTO` - the scheme is detected automatically, according to the channel labels on the device (default).

`TT_CHANNEL_NUMBER_SCHEME_ONE` - force the first channel to be 1.

`TT_CHANNEL_NUMBER_SCHEME_ZERO` - force the first channel to be 0.

Important: The method must be called before the first call to *createTimeTagger()*.

---

getTimeTaggerChannelNumberScheme()
Returns the currently used channel schema which is either `TT_CHANNEL_NUMBER_SCHEME_ZERO` or `TT_CHANNEL_NUMBER_SCHEME_ONE`.

Returns Channel schema

Return type int32

7.4 The TimeTagger class

This class provides access to the hardware and exposes methods to control hardware settings. It allows controlling the trigger levels, input delay, dead time, event filter, and test signals. Behind the scenes, it opens the USB connection, initializes the device and receives and manages the timetag stream. Every *measurement* and *virtual channel* requires a reference to the *TimeTagger* object with which it will be associated.

In TimeTagger software version 2.6.0, we have introduced the new *TimeTaggerVirtual* which allows to replay earlier stored time-tag dump files. Using virtual Time Tagger you can repeat your experiment data analysis with completely different parameters or even perform different measurements.

class TimeTagger

reset()
Reset the Time Tagger to the start-up state.

setTriggerLevel(*channel*, *voltage*)
Set the trigger level of an input channel in Volts.

Parameters

- **channel** (*int32*) – Physical channel number
- **voltage** (*double*) – Trigger level in Volts

getTriggerLevel(*channel*)
Returns trigger level for the specified physical channel number.

Parameters **channel** (*int32*) – Physical channel number

Returns The applied trigger voltage level which might differ from the input parameter due to the DAC discretization.

Return type double

setInputDelay(*channel*, *delay*)
Set the input delay compensation for the given *channel* in picoseconds. The input delay can also have a negative value.

Note: This method has the best performance with "small delays". The delay is considered "small" when less than 100 events arrive within the time of the largest delay set. For example, if the total event-rate over all channels used is 10 Mevent/s, the signal can be delayed efficiently up to 10 microseconds. For large delays, please use *DelayedChannel* instead.

Parameters

- **channel** (*int32*) – Channel number
- **delay** (*int64*) – Delay time in picoseconds

getInputDelay(*channel*)
Get the input delay compensation for the given *channel* in picoseconds.

Parameters **channel** (*int32*) – Channel number

Returns Delay time in picoseconds

Return type int64

getHardwareDelayCompensation(*channel*)
Get the hardware input delay compensation for the given *channel* in picoseconds.

The physical input delays are calibrated and compensated. However this compensation is implemented after the conditional filter and so affects its result. This function queries the effective input delay, which compensates the hardware delay.

Parameters **channel** (*int32*) – Channel number

Returns Hardware delay compensation in picoseconds

Return type int64

setConditionalFilter(*trigger*, *filtered*, *hardwareDelayCompensation*)
Activates or deactivates the event filter. Time tags on the filtered channels are discarded unless they were

preceded by a time tag on one of the trigger channels which reduces the data rate. More details can be found in the *In Depth Guide: Conditional Filter*.

Parameters

- **trigger** (*list[int32]*) – List of channel numbers
- **filtered** (*list[int32]*) – List of channel numbers
- **hardwareDelayCompensation** (*bool*) – optional, default:true. If set to false, the physical hardware delay will not be compensated. This guarantees, that the trigger tag of the conditional filter is always in before the triggered tag when the InputDelays are set to 0.

clearConditionalFilter()

Deactivates the event filter. Equivilent to setConditionalFilter({}, {}, true). Enables the physical hardware delay compensation again if it was deactivated by setConditionalfilter.

getConditionalFilterTrigger()

Returns the collection of trigger channels for the conditional filter.

Returns List of channel numbers

Return type list[int32]

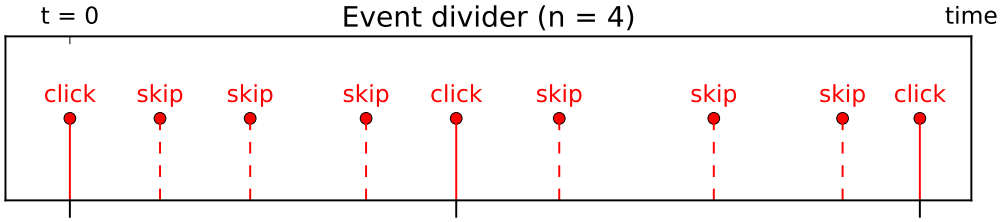
getConditionalFilterFiltered()

Returns the collection of channels to which the conditional filter is currently applied.

Returns List of channel numbers

Return type list[int32]

setEventDivider(*channel*, *divider*)



Applies an event divider filter with the specified factor to a channel, which reduces the data rate. Only every n-th event from the input stream passes through the filter, as shown in the image. Note that if the conditional filter is also active, the conditional filter is applied first.

Parameters

- **channel** (*int32*) – Physical channel number
- **divider** (*uint32*) – Divider factor.

getEventDivider(*channel*)

Gets the event divider filter factor for the given *channel*.

Parameters **channel** (*int32*) – Channel number

Returns Divider factor value

Return type uint32

setNormalization(*state*)

Enables or disables Gaussian normalization of the detection jitter. Enabled by default.

Parameters **state** (*bool*) – True/False

getNormalization()
Returns true if Gaussian normalization is enabled.

Returns True/False

Return type bool

setDeadtime(*channel*, *deadtime*)
Sets the dead time of a channel in picoseconds. The requested time will be rounded to the nearest multiple of the clock time, which is 6 ns for the Time Tagger 20 and 2 ns for the Time Tagger Ultra. The minimum dead time is one clock cycle. As the deadtime passed as an input will be altered to the rounded value, the rounded value will be returned. The maximum dead time is 393 µs for the Time Tagger 20 and 131 µs for the Time Tagger Ultra.

Parameters

- **channel** (*int32*) – Channel number.
- **deadtime** (*int64*) – Deadtime value in picoseconds.

Returns Deadtime in picoseconds rounded to the nearest valid value (multiple of the clock period not exceeding maximum dead time).

Return type int64

getDeadtime(*channel*)
Returns the dead time value for the specified *channel*.

Parameters **channel** (*int32*) – Physical channel number

Returns Deadtime value in picoseconds

Return type int64

setTestSignal(*channels*, *bool state*)
Connect or disconnect the channels with the on-chip uncorrelated signal generator.

Parameters

- **channels** (*list[int32]*) – List of physical channel numbers
- **state** (*bool*) – True/False

getTestSignal(*channel*)
Returns true if the internal test signal is activated on the specified *channel*.

Parameters **channel** (*int32*) – Physical channel number

Returns True/False

Return type bool

getSerial()
Returns the hardware serial number.

Returns Serial number string

Return type string

getOverflows()
Returns the number of overflows (missing blocks of time tags due to limited USB data rate) that occurred since start-up or last call to *clearOverflows()*.

Returns Number of overflows

Return type int64

getOverflowsAndClear()
Returns the number of overflows that occurred since start-up and sets them to zero (see, *clearOverflows()*).

Returns Number of overflows

Return type int64

clearOverflows()
Set the overflow counter to zero.

getFence(*alloc_fence*)
Generate a new fence object, which validates the current configuration and the current time. This fence is uploaded to the earliest pipeline stage of the Time Tagger. Waiting on this fence ensures that all hardware settings such as trigger levels, channel registrations, etc., have propagated to the FPGA and are physically active. Synchronizes the Time Tagger internal memory, so that all tags arriving after the waitForFence call were actually produced after the getFence call. The waitForFence function waits until all tags, which are present at the time of the function call within the internal memory of the Time Tagger, are processed. This call might block to limit the amount of active fences.

Parameters **alloc_fence** (*bool*) – optional, default: true. If false, a reference to the most recently created fence will be returned instead

Returns The allocated fence

Return type int32

waitForFence(*fence*, *timeout*)
Wait for a fence in the data stream. See *getFence()* for more details.

Parameters

- **fence** (*int32*) – fence object, which shall be waited on
- **timeout** (*int32*) – optional, default: -1. timeout in milliseconds. Negative means no timeout, zero returns immediately.

Returns True if the fence has passed, false on timeout

Return type bool

sync()
Ensure that all hardware settings such as trigger levels, channel registrations, etc., have propagated to the FPGA and are physically active. Synchronizes the Time Tagger internal memory, so that all tags arriving after a sync call were actually produced after the sync call. The sync function waits until all tags, which are present at the time of the function call within the internal memory of the Time Tagger, are processed.

getPcbVersion()
Returns Time Tagger PCB (Printed circuit board) version.

Returns PCB version

Return type string

getDACRange()
Return a vector containing the minimum and the maximum DAC (Digital-to-Analog Convertor) voltage range for the trigger level.

Returns Min and max voltage in Volt

Return type (double, double)

registerChannel(*channel*)
Enable transmission of time tags on the specified channel.

Parameters **channel** (*int32*) – Channel number

unregisterChannel(*channel*)
Disable transmission of time tags on the specified channel.

Parameters **channel** (*int32*) – Channel number

getChannelList(*type*)
Returns a list of channels. The parameter *type* can be one of the following values:

TT_CHANNEL_RISING_AND_FALLING_EDGES all channels, both rising and falling edges (default)
TT_CHANNEL_RISING_EDGES the channels of the rising edges
TT_CHANNEL_FALLING_EDGES the channels of the falling edges

Parameters **type** (*int*) – Defines what channels to be returned

Returns List of channel numbers

Return type list[int32]

getInvertedChannel(*channel*)
Returns the channel number for the inverted edge of the channel passed in via the channel parameter. In case the given channel has no inverted channel, *CHANNEL_UNUSED* is returned.

Parameters **channel** (*int32*) – Channel number

Returns Channel number

Return type int32

isChannelUnused(*channel*)
Returns true if the passed channel number is *CHANNEL_UNUSED*.

Parameters **channel** (*int32*) – Channel number

Returns True/False

Return type bool

setHardwareBufferSize(size) - TT Ultra only
Sets the maximum buffer size within the Time Tagger Ultra. The default value is 64 MTags, but can be changed within the range of 32 kTags to 512 MTags. Please note that this buffer can only be filled with a total data rate of up to 500 MTags/s.

Parameters **size** (*int32*) – Buffer size, must be a positive number

autoCalibration()
Run an auto-calibration of the Time Tagger hardware using the built-in test signal.

Returns the list of jitter of each input channel in ps based on the calibration data.

Return type list[double]

getDistributionCount()
Returns the calibration data represented in counts.

Returns Distribution data

Return type 2d_array[int64]

getDistributionPSec()
Returns the calibration data in picoseconds.

Returns Calibration data

Return type 2d_array[int64]

getPsPerClock()
Returns the duration of a clock cycle in picoseconds. This is the inverse of the internal clock frequency.

Returns Clock period in picoseconds

Return type int64

setStreamBlockSize(*max_events*, *max_latency*)
This option controls the latency and the block size of the data stream. The default values are max_events = 131072 events and max_latency = 20 ms. Depending on which of the two parameters is exceeded first, the block stream size is adjusted accordingly. The block size will be reduced automatically for blocks when no signal is arriving for 512 ns on the Time Tagger Ultra and 1536 ns for the Time Tagger 20.

setTestSignalDivider(*divider*)
Change the frequency of the on-chip test signal, the default value 63 corresponds to ~800 kCounts/s.

Parameters **divider** (*int32*) – Division factor

getTestSignalDivider()
Returns the value of test signal division factor.

getSensorData() - TT Ultra only
Prints all available sensor data for the given board.

Returns Tabulated sensor data

Return type string

setLED(*bitmask*)
Manually change the state of the Time Tagger LEDs. The power LED of the Time Tagger 20 cannot be programmed by software.

Example:

```
# Turn off all LEDs
tagger.setLED(0x01FF0000)

# Restore normal LEDs operation
tagger.setLED(0)
```

0 -> LED off
1 -> LED on
illumination bits
0-2: status, rgb - all Time Tagger models
3-5: power, rgb - Time Tagger Ultra only
6-8: clock, rgb - Time Tagger Ultra only

0 -> normal LED behavior, not overwritten by setLED
1 -> LED state is overwritten by the corresponding bit of 0-8
mask bits
16-18: status, rgb - all Time Tagger models
19-21: power, rgb - Time Tagger Ultra only
22-24: clock, rgb - Time Tagger Ultra only

Parameters **bitmask** (*uint32*) – LED bitmask.

getConfiguration()
Returns a JSON formatted string containing a complete information on the Time Tagger settings.

7.5 The TimeTaggerVirtual class

In the TimeTagger software version 2.6.0, we have introduced the new *TimeTaggerVirtual* which allows to replay earlier stored time-tag dump files. Using the virtual Time Tagger you can repeat your experiment data analysis with completely different parameters or even perform different measurements.

Note: The virtual Time Tagger requires a free software license, which is automatically acquired from the Swabian Instruments license server when .createTimeTagger() or .createTimeTaggerVirtual() is called while a Time Tagger is attached. Once the license is received, it is permanently stored on this PC. The virtual Time Tagger can be used offline afterward without having a Time Tagger attached.

---

class TimeTaggerVirtual

replay(*file*[, *begin=0*, *duration=-1*, *queue=True*])
Replay a dump file specified by its path *file*.

This method adds the file to the replay queue. If the flag 'queue' is false, the current queue will be flushed and this file will be replayed immediatelly.

Parameters

- **file** (*string*) – the file to be replayed
- **begin** (*int64*) – duration in ps to skip at the beginnning of the file. A negative time will generate a pause in the replay.
- **duration** (*int64*) – duration in ps to be read from the file. *duration=-1* will replay everything. (default: -1)
- **queue** (*bool*) – flag if this file shall be queued. (default: *True*)

Returns ID of the queued file

Return type uint64

stop()
This method stops the current file and clears the replay queue.

waitForCompletion([*ID=0*, *timeout=-1*])
Blocks the current thread until the replay is completed.

This method blocks the current execution and waits until the given file has finished its replay. If no ID is provided, it waits until all queued files are replayed.

This function does not block on a zero timeout. Negative timeouts are interpreted as infinite timeouts.

Parameters

- **ID** (*uint64*) – selects which file to wait for. (default: 0)
- **timeout** (*int*) – timeout in milliseconds

Returns true if the file is complete, false on timeout

Return type bool

setReplaySpeed(*speed*)
Configures the speed factor for the virtual tagger.

A value of *speed=1.0* will replay at a real time rate. All *speed* values < 0.0 will replay the data as fast as possible, but stops at the end of all data. This is the default value. Extrem slow replay speed between 0.0 and 0.1 is not supported.

Parameters **speed** (*double*) – replay speed factor.

getReplaySpeed()
Returns the current speed factor.

Please see also *setReplaySpeed()* for more details.

7.6 Virtual Channels

Virtual channels are software-defined channels as compared to the real input channels. Virtual channels can be understood as a stream flow processing units. They have an input through which receive timetags from a real or another virtual channel and output to which they send processed timetags.

Virtual channels are used as input channels to the measurement classes the same way as real channels. Since the virtual channels are created during run-time, the corresponding channel number(s) are assigned dynamically and can be retrieved using *getChannel()* or *getChannels()* methods of virtual channel object.

7.6.1 Available virtual channels

Note: In MATLAB, the Virtual Channel names have common prefix `TT*`. For example: `Combiner` is named as `TTCombiner`. This prevents possible name collisions with existing MATLAB or user functions.

Combiner Combines two or more channels into one.

ConstantFractionDiscriminator Detects rising and falling edges of an input pulse and returns the average time.

Coincidence Detects coincidence clicks on two or more channels within a given window.

Coincidences Detects coincidence clicks on multiple channel groups within a given window.

DelayedChannel Clones input channels which can be delayed.

FrequencyMultiplier Frequency Multiplier for a channel with a periodic signal.

GatedChannel Transmits signals of an input_channel depending on the signals arriving at gate_start_channel and gate_stop_channel.

EventGenerator Generates a signal pattern for every trigger signal.

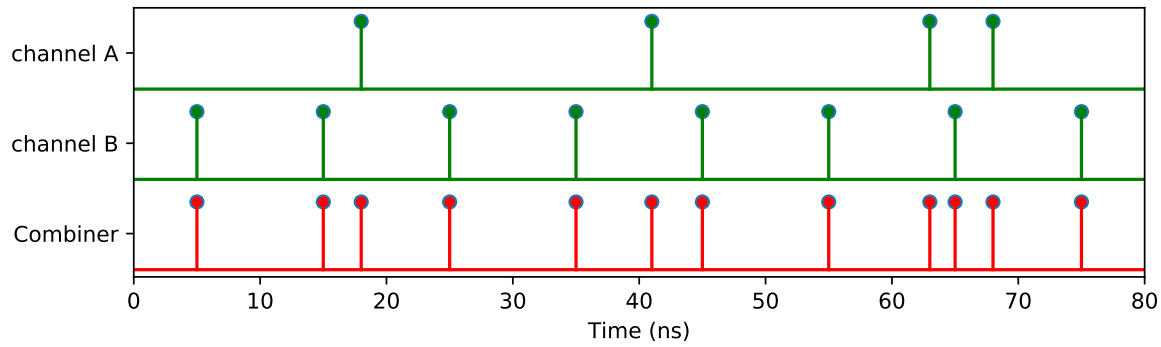
7.6.2 Common methods

`VirtualChannel.`**`getChannel`**`()`
`VirtualChannel.`**`getChannels`**`()`

Returns the channel number(s) corresponding to the virtual channel(s). Use this channel number the very same way as the channel number of physical channel, for example, as an input to a measurement class or another virtual channel.

Important: Virtual channels operate on the time tags that arrive at their input. These time tags can be from rising or falling edges of the physical signal. However, the virtual channels themselves do not support such a concept as an inverted channel.

7.6.3 Combiner



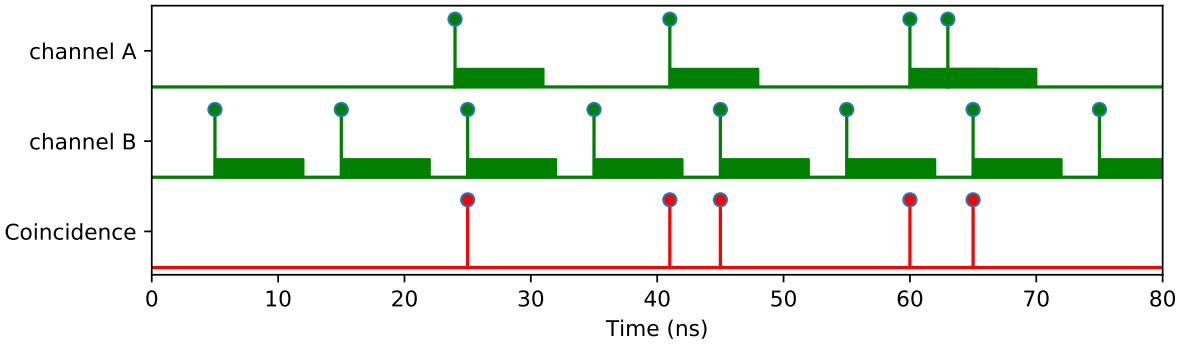
Combines two or more channels into one. The virtual channel is triggered, e.g., for two channels when either channel A OR channel B received a signal.

`class Combiner`(*tagger*, *channels=[]*)

Parameters

- **`tagger`** (`TimeTagger`) – time tagger object instance
- **`channels`** (*`list[int32]`*) – List of channels to be combined into a single virtual channel

7.6.4 Coincidence



Detects coincidence clicks on two or more channels within a given window. The virtual channel is triggered, e.g., when channel A AND channel B received a signal within the given coincidence window. The timestamp of the coincidence on the virtual channel is the time of the last event arriving to complete the coincidence.

class Coincidence(*tagger*, *channels*, *coincidenceWindow*, *timestamp*)

Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **channels** (*list[int32]*) – list of channels on which coincidence will be detected in the virtual channel
- **coincidenceWindow** (*int64*) – maximum time between all events for a coincidence [ps]
- **timestamp** (*CoincidenceTimestamp*) – type of timestamp for virtual channel (Last, Average, First, ListedFirst)

Coincidence Timestamp Type:

- **Last** - time of the last event completing the coincidence (fastest option - default)
- **Average** - average time of all tags completing the coincidence
- **First** - time of the first event received of the coincidence
- **ListedFirst** - time of the first channel of the list with which the Coincidence was initialized

The CoincidenceTimestamp is an enum. It is accessable in all languages, in Phyton via *TimeTagger.CoincidenceTimestamp* and Matlab via *TTCoincidenceTimestamp*.

7.6.5 Coincidences

Detects coincidence clicks on multiple channel groups within a given window. If several different coincidences are required with the same window size, *Coincidences* provides better performance in comparison to multiple virtual *Coincidence* channels.

Example code:

```
coinc = Coincidences(tagger, [[1,2], [2,3,5]], coincidenceWindow=10000)
coinc_chans = coinc.getChannels()
coinc1_ch = coinc_chans[0]  # double coincidence in channels [1,2]
coinc2_ch = coinc_chans[1]  # triple coincidence in channels [2,3,5]
```

(continues on next page)

(continued from previous page)

```
# or equivalent but less performant
coinc1 = Coincidence(tagger, [1,2], coincidenceWindow=10000)
coinc2 = Coincidence(tagger, [2,3,5], coincidenceWindow=10000)
coinc1_ch = coinc1.getChannel()  # double coincidence in channels [1,2]
coinc2_ch = coinc2.getChannel()  # triple coincidence in channels [2,3,5]
```

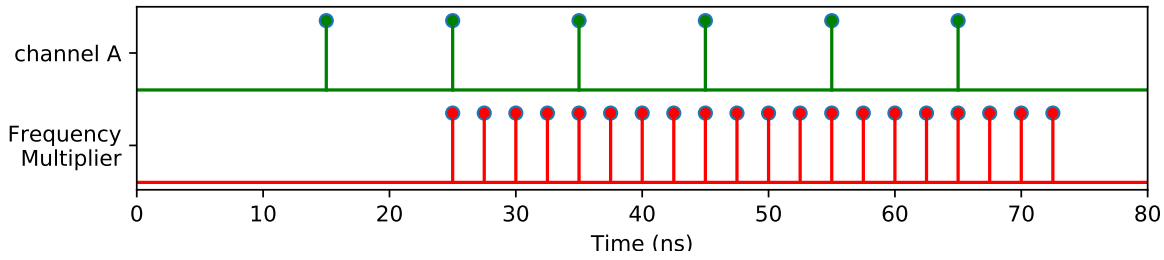
Note: Only C++ and python support jagged arrays (array of arrays, like uint[][]) which are required to combine several coincidence groups and pass them to the constructor of the Coincidences class. Hence, the API differs for Matlab, which requires a cell array of 1D vectors to be passed to the constructor (see Matlab examples provided with the installer). For LabVIEW, a CoincidencesFactory-Class is available to create a Coincidences object, which is also shown in the LabVIEW examples provided with the installer).

class Coincidences (*tagger*, *coincidenceGroups*, *coincidenceWindow*, *timestamp*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **coincidenceGroups** (*list[list[int32]]*) – list of channel groups on which coincidence will be detected in the virtual channel
- **coincidenceWindow** (*int64*) – maximum time between all events for a coincidence [ps]
- **timestamp** (*CoincidenceTimestamp*) – type of timestamp for virtual channel (Last, Average, First, ListedFirst)

7.6.6 FrequencyMultiplier



Frequency Multiplier for a channel with a periodic signal.

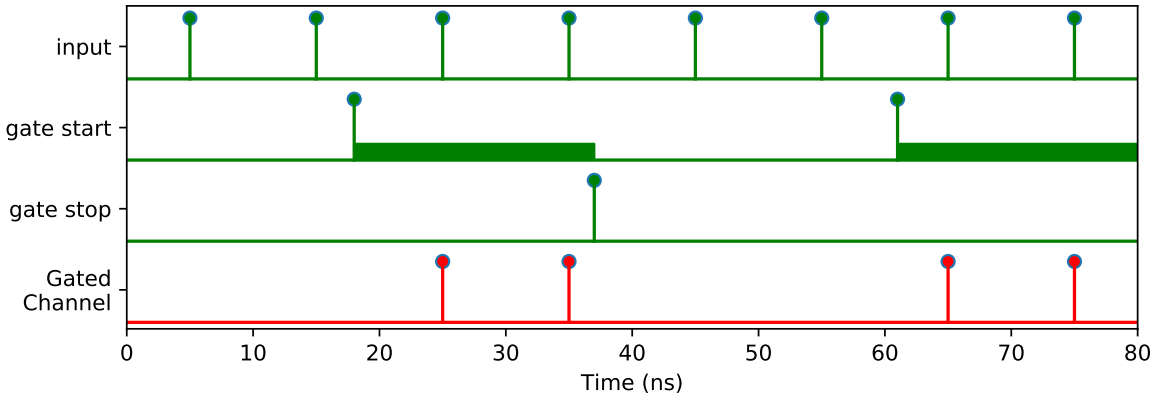
Note: Very high output frequencies create a high CPU load, eventually leading to *overflows*.

class FrequencyMultiplier (*tagger*, *input_channel*, *multiplier*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **input_channel** (*int32*) – channel on which the upscaling of the frequency is based on
- **multiplier** (*int32*) – frequency upscaling factor

7.6.7 GatedChannel



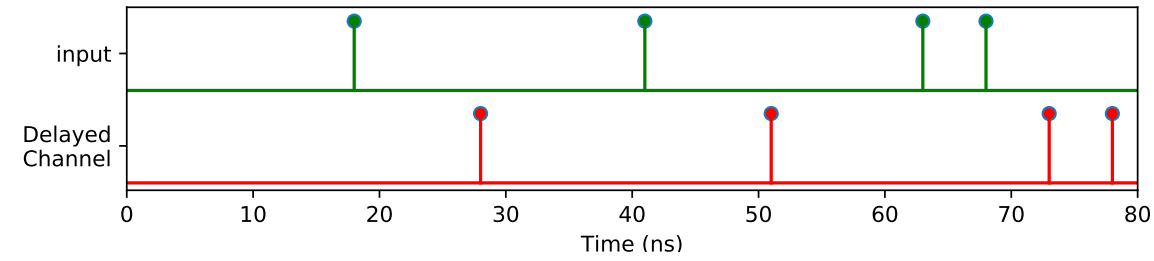
Transmits the signal from an input_channel to a new virtual channel between an edge detected at the *gate_start_channel* and the *gate_stop_channel*.

class GatedChannel(*tagger*, *input_channel*, *gate_start_channel*, *gate_stop_channel*)

Parameters

- **tagger** (*TimeTagger*) – time tagger object
- **input_channel** (*int32*) – channel which is gated
- **gate_start_channel** (*int32*) – channel on which a signal detected will start the transmission of the input_channel through the gate
- **gate_stop_channel** (*int32*) – channel on which a signal detected will stop the transmission of the input_channel through the gate

7.6.8 DelayedChannel



Clones input channels, which can be delayed by a time specified with the *delay* parameter in the constructor or the *setDelay()* method. A negative delay will delay all other events.

Note: If you want to set a global delay for one or more input channels, *TimeTagger.setInputDelay()* is recommended as long as the delays are small, which means that not more than 100 events on all channels should arrive within the maximum delay set.

---

class DelayedChannel(*tagger*, *input_channel*, *delay*)

Parameters

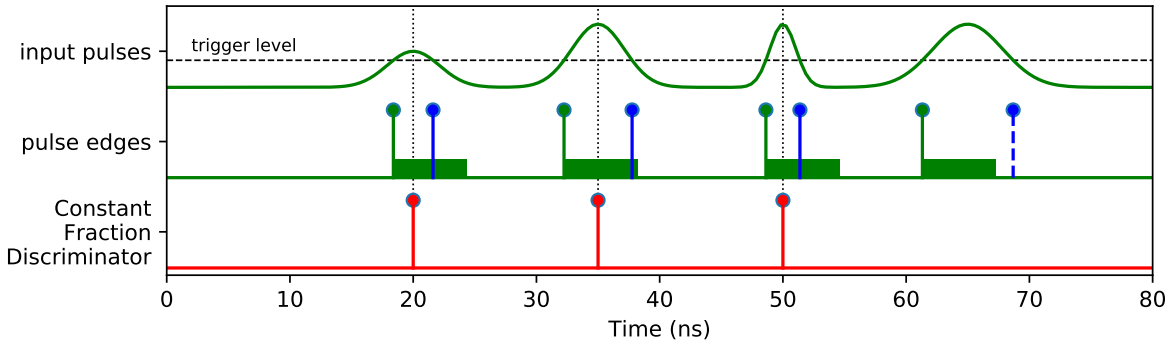
- **tagger** (*TimeTagger*) – time tagger object
- **input_channel** (*int32*) – channel to be delayed
- **delay** (*int64*) – amount of time to delay in ps

setDelay (*delay*)
Allows modifying the delay time.

> **Warning:** Calling this method with a reduced delay time may result in a partial loss of the internally buffered time tags.

Parameters **delay** (*int64*) – Delay time in picoseconds

7.6.9 ConstantFractionDiscriminator



Constant Fraction Discriminator (CFD) detects rising and falling edges of an input pulse and returns the average time of both edges. This is useful in situations when precise timing of the pulse position is desired for the pulses of varying durations and amplitudes.

For example, the figure above shows four input pulses separated by 15 nanoseconds. The first two pulses have equal widths but different amplitudes, the middle two pulses have equal amplitude but different durations, and the last pulse has a duration longer than the *search_window* and is therefore skipped. For such input signal, if we measure the time of the rising edges only, we get an error in the pulse positions, while with CFD this error is eliminated for symmetric pulses.

Note: The virtual CFD requires the time tags of the rising and falling edge. Hence, the transferred data of the input channel is twice the regular input rate.

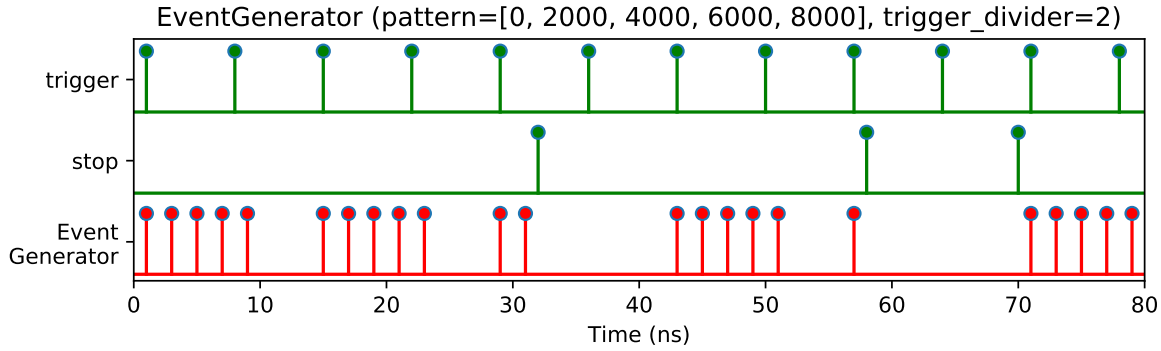
---

class ConstantFractionDiscriminator (*tagger*, *channels*, *search_window*)

Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **channels** (*list[int32]*) – list of channels on which to perform CFD
- **search_window** (*int64*) – max pulse duration in picoseconds to be detected

7.6.10 EventGenerator



Emits an arbitrary pattern of timestamps for every trigger event. The number of trigger events can be reduced by *trigger_divider*. The start of a new pattern does not abort the execution of unfinished patterns, so patterns may overlap. The execution of all running patterns can be aborted by a click of the *stop_channel*, i.e. overlapping patterns can be avoided by setting the *stop_channel* to the *trigger_channel*.

class EventGenerator(*tagger*, *trigger_channel*, *pattern*, *trigger_divider*, *stop_channel*)

Parameters

- **tagger** (*TimeTagger*) – Time Tagger object instance.
- **trigger_channel** (*int32*) – Channel number of the trigger signal.
- **pattern** (*list[int64]*) – List of relative timestamps defining the pattern executed upon a trigger event.
- **trigger_divider** (*uint64*) – Factor by which the number of trigger events is reduced. (default: 1)
- **divider_offset** (*uint64*) – If *trigger_divider* > 1, the *divider_offset* the number of trigger clicks to be ignored before emitting the first pattern. (default: 0)
- **stop_channel** (*int32*) – Channel number of the stop channel. (optional)

7.7 Measurement Classes

The Time Tagger library includes several classes that implement various measurements. All measurements are derived from a base class called 'IteratorBase' that is described further down. As the name suggests, it uses the *iterator* programming concept.

All measurements provide a small number of methods to start and stop the execution and to access the accumulated data.

7.7.1 Available measurement classes

Note: In MATLAB, the Measurement names have common prefix `TT*`. For example: `Correlation` is named as `TTCorrelation`. This prevents possible name collisions with existing MATLAB or user functions.

Correlation Auto- and Cross-correlation measurement.

CountBetweenMarkers Counts tags on one channel within bins which are determined by triggers on one or two other channels. Uses a static buffer output. Use this to implement a gated counter, a counter synchronized to external signals, etc.

Counter Counts the clicks on one or more channels with a fixed bin width and a circular buffer output.

Countrate Average tag rate on one or more channels.

FLIM Fluorescence lifetime imaging.

IteratorBase Base class for implementing custom measurements (only C++ and Python).

Histogram A simple histogram of time differences. This can be used to measure lifetime, for example.

Histogram2D A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectrocopy.

HistogramLogBins Accumulates time differences into a histogram with logarithmic increasing bin sizes.

Scope Detects the rising and falling edges on a channel to visualize the incoming signals similar to an ultrafast logic analyzer.

StartStop Accumulates a histogram of time differences between pairs of tags on two channels. Only the first stop tag after a start tag is considered. Subsequent stop tags are discarded. The histogram length is unlimited. Bins and counts are stored in an array of tuples.

TimeDifferences Accumulates the time differences between tags on two channels in one or more histograms. The sweeping through of histograms is optionally controlled by one or two additional triggers.

TimeDifferencesND A multidimensional implementation of the TimeDifferences measurement for asynchronous next histogram triggers.

SynchronizedMeasurements Helper class that allows synchronization of the measurement classes.

Dump Deprecated - please use *FileWriter* instead. Dump measurement writes all time-tags into a file.

TimeTagStream This class provides you with access to the time-tag stream and allows you to implement your own on-the-fly processing.

FileWriter This class writes time-tags into a file with a lossless compression and is replaces the *Dump* class.

FileReader Allows you to read time-tags from a file written by the *FileWriter* and provides data in the same format as the`TimetagStream`_.

7.7.2 Common methods

getData()
Returns a copy of data currently present in data buffer. The returned data can be a scalar, single- or multi-dimensional array, depending on the specific measurement class. This method does not disturb running measurement and can be safely used to get intermediate results.

clear()
Discards accumulated measurement data and initializes the data buffer with zero values.

start()
Starts or continues data acquisition. This method is implicitly called when a measurement object is created.

startFor(*duration*[, *clear=True*])
Starts or continues the data acquisition for the given duration (in ps). After the *duration* time, the method `stop()` is called and `isRunning()` will return False. Whether the accumulated data is cleared at the beginning of `startFor()` is controlled with the second parameter *clear*, which is True by default.

stop()
After calling this method, the measurement will stop processing incoming tags. Use `start()` or `startFor()` to continue or restart the measurement.

isRunning()
Returns True if the measurement is collecting the data. This method will returns False if the measurement was stopped manually by calling `stop()` or automatically after calling `startFor()` and the *duration* has passed.

Note: All measurements start accumulating data immediately after their creation.

Returns True/False

Return type bool

getCaptureDuration()
Total capture duration since the measurement creation or last call to `clear()`.

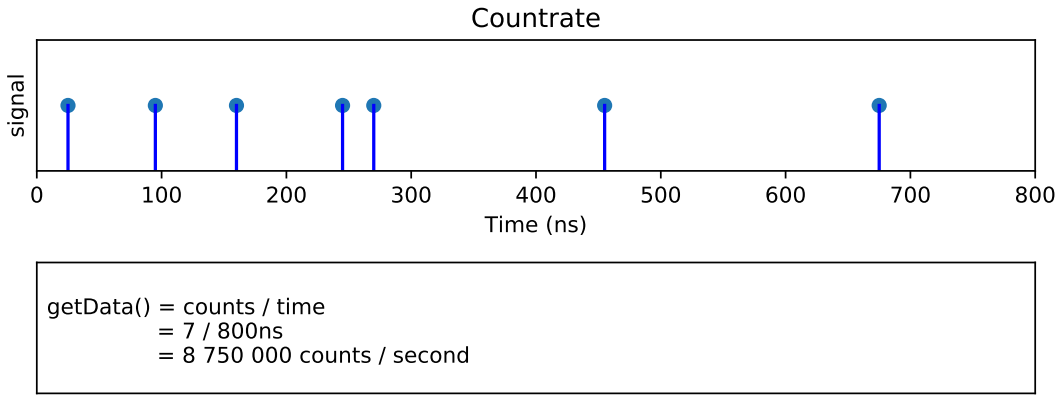
In a typical application, the following steps are performed (see *example*):

1. Create an instance of a measurement
2. Wait for some time
3. Retrieve the data accumulated by the measurement by calling the `getData()` method.

The specific measurements are described below.

7.7.3 Event counting

Countrate



Measures the average count rate on one or more channels. Specifically, it determines the counts per second on the specified channels starting from the very first tag arriving after the instantiation or last call to `clear()` of the measurement.

class Countrate(*tagger*, *channels*)

Parameters

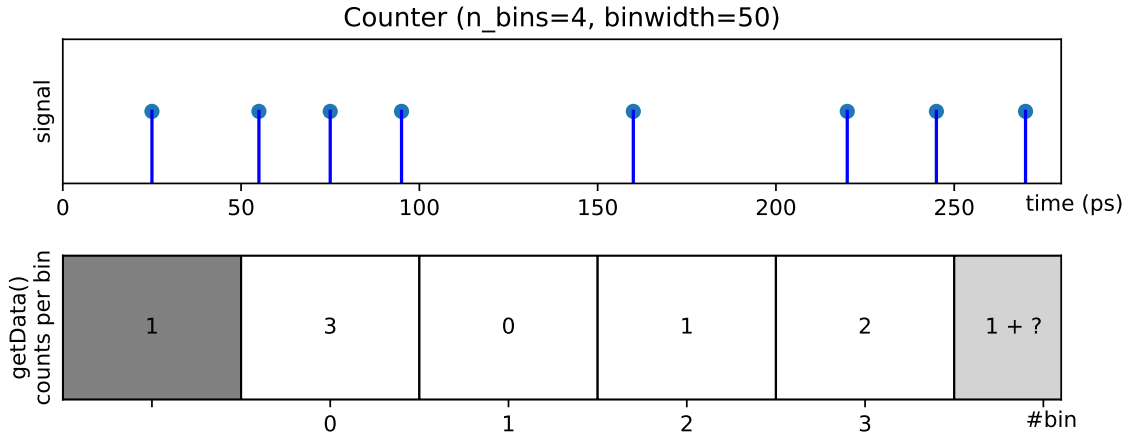
- **tagger** (TimeTagger) – time tagger object instance
- **channels** (*list[int32]*) – channels for which the average count rate is measured

getData()
Returns the average count rate in counts per second.

getCountsTotal()
Returns the total amount of events since the instantiation of this object.

clear()
Resets the accumulated counts to zero and restarts the measurement with the next incoming tag.

Counter



Time trace of the count rate on one or more channels. Specifically, this measurement repeatedly counts tags on one or more channels within a time interval *binwidth* and stores the results in a two-dimensional array of size 'number of channels' by 'n_values'. The array is treated as a circular buffer, which means all values in the array are shifted by one position when a new value is generated. The last entry in the array is always the most recent value.

class Counter(*tagger*, *channels*, *binwidth*, *n_values*)

Parameters

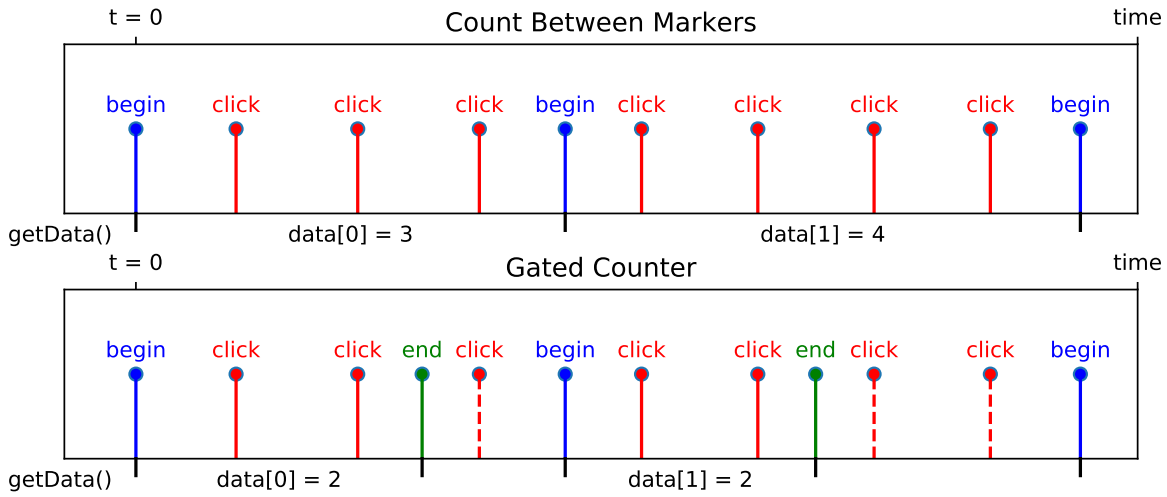
- **tagger** (TimeTagger) – time tagger object
- **channels** (*list[int32]*) – channels used for counting tags
- **binwidth** (*int64*) – bin width in ps
- **n_values** (*uint32*) – number of bins (data buffer size)

getData()
Returns an array of size 'number of channels' by *n_values* containing the current values of the circular buffer (counts in each bin).

getIndex()
Returns a vector of size *n_values* containing the time bins in ps.

clear()
Resets the array to zero and restarts the measurement.

CountBetweenMarkers



Counts events on a single channel within the time indicated by a "start" and "stop" signals. The bin edges between which counts are accumulated are determined by one or more hardware triggers. Specifically, the measurement records data into a vector of length *n_values* (initially filled with zeros). It waits for tags on the *begin_channel*. When a tag is detected on the *begin_channel* it starts counting tags on the *click_channel*. When the next tag is detected on the *begin_channel* it stores the current counter value as the next entry in the data vector, resets the counter to zero and starts accumulating counts again. If an *end_channel* is specified, the measurement stores the current counter value and resets the counter when a tag is detected on the *end_channel* rather than the *begin_channel*. You can use this, e.g., to accumulate counts within a gate by using rising edges on one channel as the *begin_channel* and falling edges on the same channel as the *end_channel*. The accumulation time for each value can be accessed via *getBinWidths()*. The measurement stops when all entries in the data vector are filled.

class CountBetweenMarkers(*tagger*, *click_channel*, *begin_channel*, *end_channel*, *n_values*)

Parameters

- **tagger** (TimeTagger) – time tagger object
- **click_channel** (*int32*) – channel on which clicks are received, gated by begin_channel and end_channel
- **begin_channel** (*int32*) – channel that triggers the beginning of counting and stepping to the next value
- **end_channel** (*int32*) – channel that triggers the end of counting
- **n_values** (*uint32*) – number of values stored (data buffer size)

getData()
Returns an array of size *n_values* containing the acquired counter values.

getIndex()
Returns a vector of size *n_values* containing the time in ps of each start click in respect to the very first start click.

getBinWidths()
Returns a vector of size *n_values* containing the time differences of 'start -> (next start or stop)' for the acquired counter values.

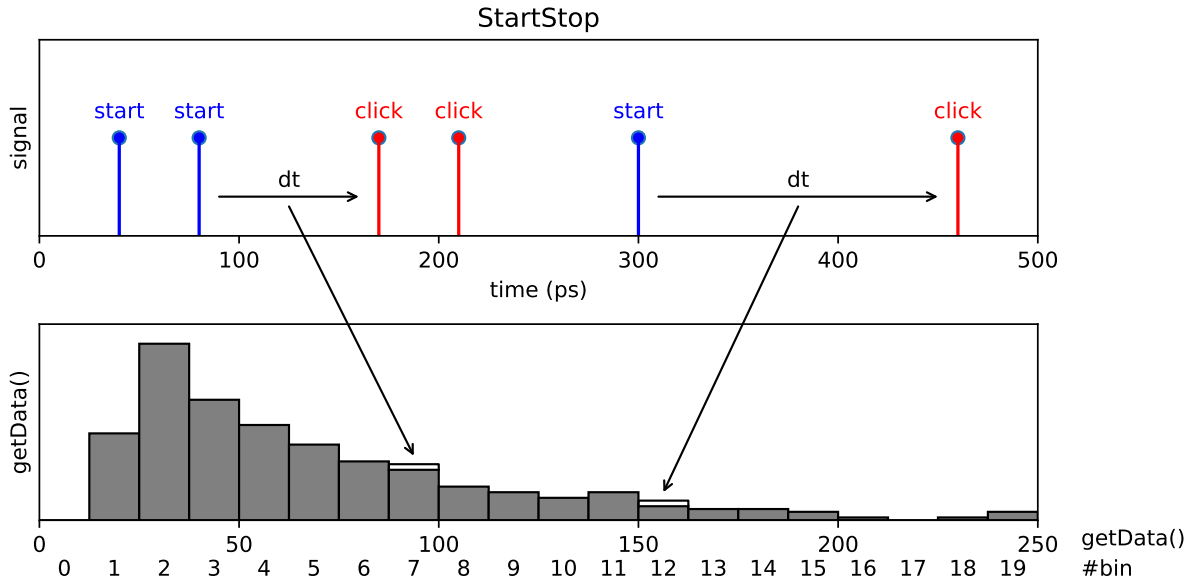
clear()
Resets the array to zero and restarts the measurement.

ready()
Returns True when the entire array is filled.

7.7.4 Time histograms

This section describes various measurements that calculate time differences between events and accumulate the results into a histogram.

StartStop



A simple start-stop measurement. This class performs a start-stop measurement between two channels and stores the time differences in a histogram. The histogram resolution is specified beforehand (*binwidth*) but the histogram range (number of bins) is unlimited. It is adapted to the largest time difference that was detected. Thus, all pairs of subsequent clicks are registered. Only non-empty bins are recorded.

class StartStop(*tagger*, *click_channel*, *start_channel*, *binwidth*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which stop clicks are received
- **start_channel** (*int32*) – channel on which start clicks are received
- **binwidth** (*int64*) – bin width in ps

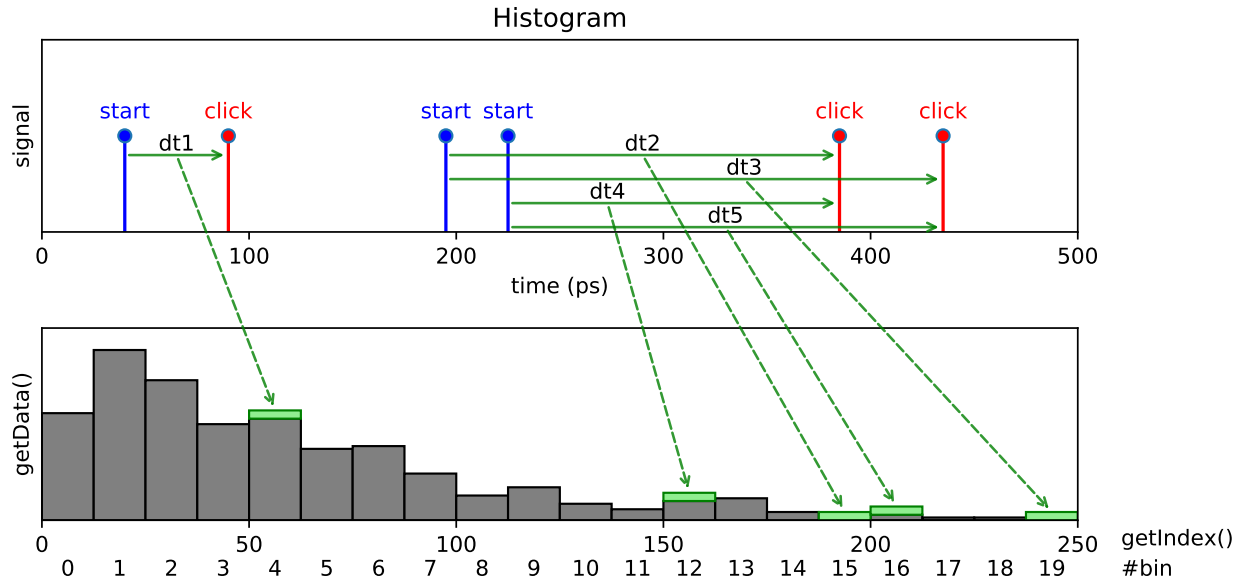
getData()

Returns an array of tuples (array of shape Nx2) containing the times (in ps) and counts of each bin. Only non-empty bins are returned.

clear()

Resets the array to zero and restarts the measurement.

Histogram



Accumulate time differences into a histogram. This is a simple multiple start, multiple stop measurement. This is a special case of the more general *TimeDifferences* measurement. Specifically, the measurement waits for clicks on the *start_channel*, and for each start click, it measures the time difference between the start clicks and all subsequent clicks on the *click_channel* and stores them in a histogram. The histogram range and resolution are specified by the number of bins and the bin width specified in ps. Clicks that fall outside the histogram range are ignored. Data accumulation is performed independently for all start clicks. This type of measurement is frequently referred to as a 'multiple start, multiple stop' measurement and corresponds to a full auto- or cross-correlation measurement.

class Histogram(*tagger*, *click_channel*, *start_channel*, *binwidth*, *n_bins*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which clicks are received
- **start_channel** (*int32*) – channel on which start clicks are received
- **binwidth** (*int64*) – bin width in ps
- **n_bins** (*int32*) – the number of bins in the histogram

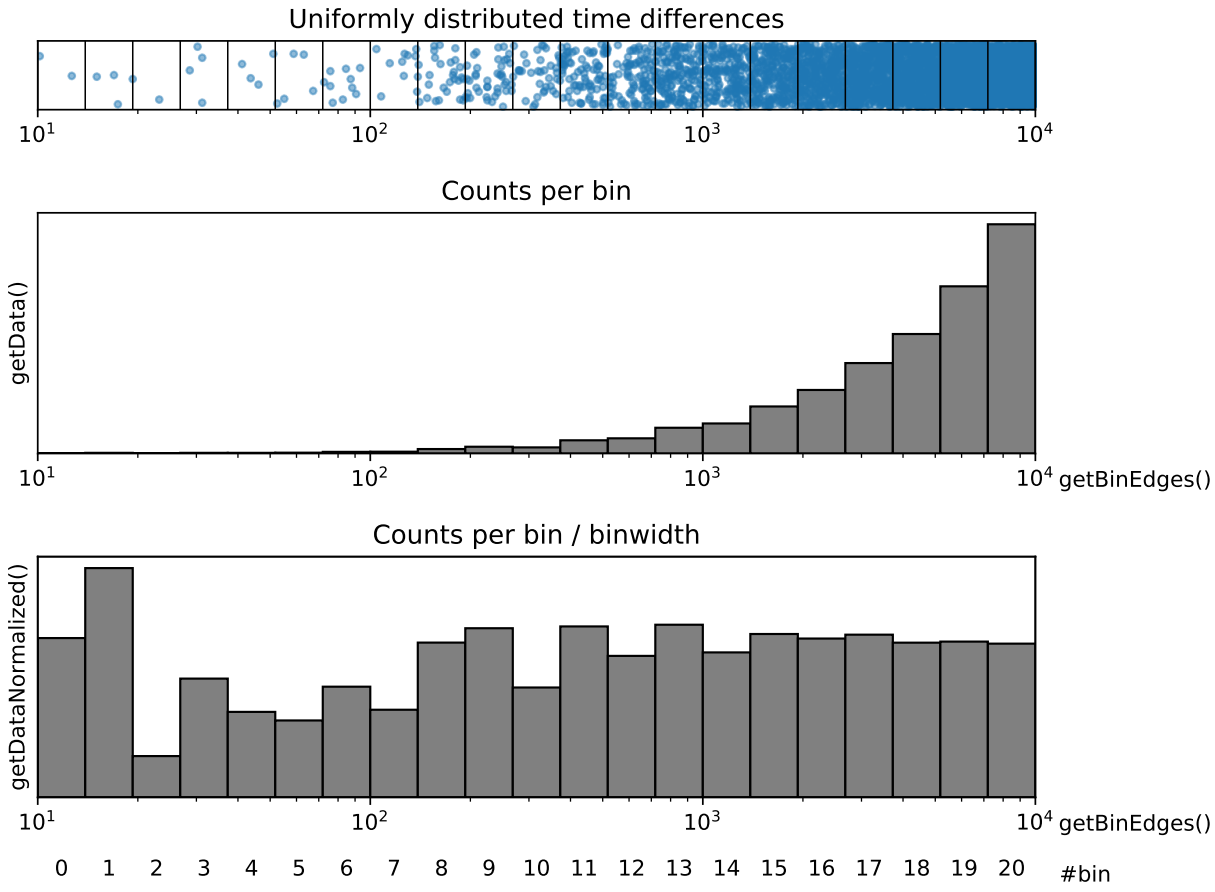
getData()
Returns a one-dimensional array of size *n_bins* containing the histogram.

getIndex()
Returns a vector of size *n_bins* containing the time bins in ps.

clear()
Resets the array to zero.

HistogramLogBins

The HistogramLogBins measurement is similar to *Histogram* but the bin widths are spaced logarithmically.



class HistogramLogBins

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which clicks are received
- **start_channel** (*int32*) – channel on which start clicks are received
- **exp_start** (*float*) – exponent `10^exp_start` in seconds where the very first bin begins
- **exp_stop** (*float*) – exponent `10^exp_stop` in seconds where the very last bin ends
- **n_bins** (*int32*) – the number of bins in the histogram

getData()
Returns a one-dimensional array of size *n_bins* containing the histogram.

getDataNormalizedCountsPerPs()
Returns the counts normalized by the binwidth of each bin.

getDataNormalizedG2()
Returns the counts normalized by the binwidth of each bin and the average count rate. This matches the implementation of *Correlation.getDataNormalized()*.

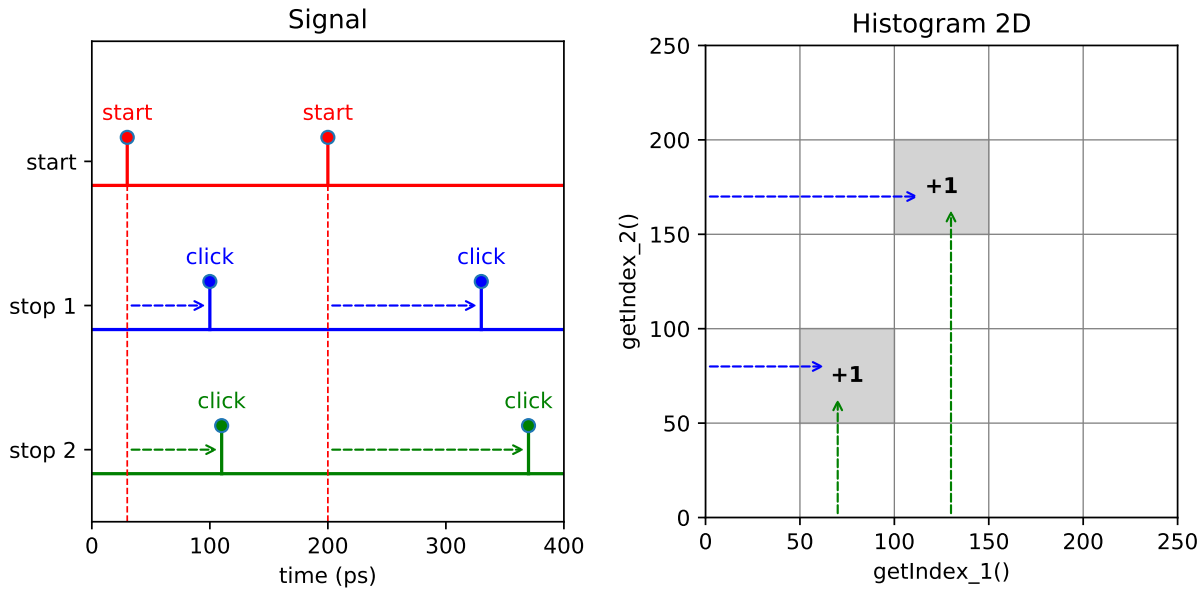
$$g^{(2)}(\tau) = \frac{\Delta t}{binwidth(\tau) \cdot N_1 \cdot N_2} \cdot histogram(\tau)$$

where Δt is the capture duration, N_1 and N_2 are number of events in each channel.

getBinEdges()
Returns a vector of size *n_bins+1* containing the bin edges in picoseconds.

clear()
Resets the array to zero.

Histogram2D



This measurement is a 2-dimensional version of the *Histogram* measurement. The measurement accumulates two-dimensional histogram where stop signals from two separate channels define the bin coordinate. For instance, this kind of measurement is similar to that of typical 2D NMR spectroscopy. The data within the histogram is aquired via a single-start, single-stop analysis for each axis.

class Histogram2D(*tagger*, *start_channel*, *stop_channel_1*, *stop_channel_2*, *binwidth_1*, *binwidth_2*, *n_bins_1*, *n_bins_2*)

Parameters

- **tagger** (TimeTagger) – time tagger object
- **start_channel** (*int32*) – channel on which start clicks are received
- **stop_channel_1** (*int32*) – channel on which stop clicks for the time axis 1 are received
- **stop_channel_2** (*int32*) – channel on which stop clicks for the time axis 2 are received
- **binwidth_1** (*uint64*) – bin width in ps for the time axis 1

- **binwidth_2** (*uint64*) – bin width in ps for the time axis 2
- **n_bins_1** (*uint*) – the number of bins along the time axis 1
- **n_bins_2** (*uint*) – the number of bins along the time axis 2

getData()
Returns a two-dimensional array of size *n_bins_1* by *n_bins_2* containing the 2D histogram.

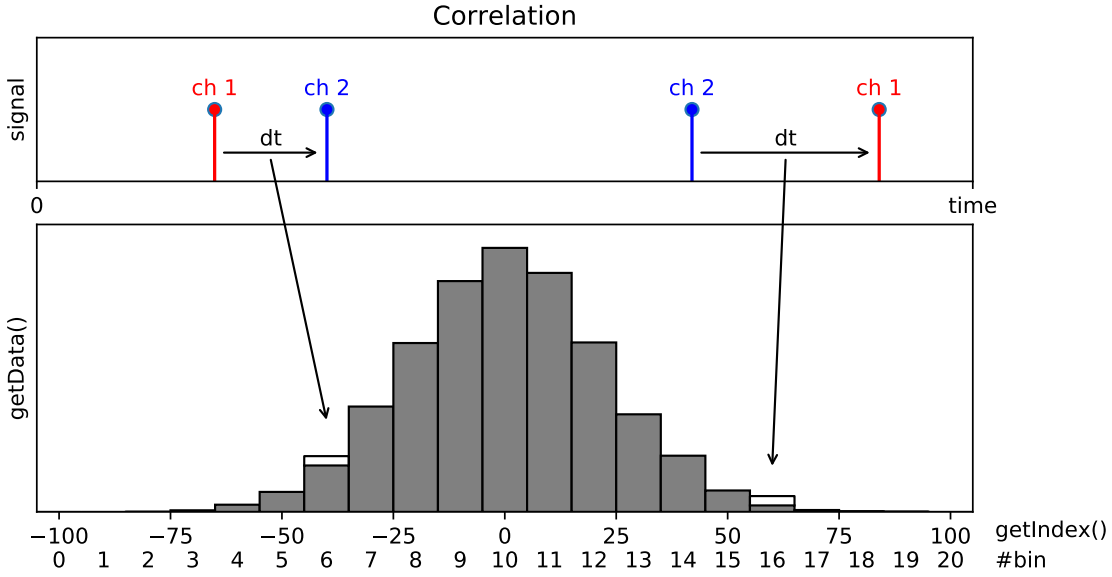
getIndex()
Returns a 3D array containing two coordinate matrices (*meshgrid*) for time bins in ps for the time axes 1 and 2. For details on *meshgrid* please take a look at the respective documentation either for Matlab or Python NumPy.

getIndex_1()
Returns a vector of size *n_bins_1* containing the bin locations in ps for the time axis 1.

getIndex_2()
Returns a vector of size *n_bins_2* containing the bin locations in ps for the time axis 2.

clear()
Resets the accumulated data.

Correlation



Accumulates time differences between clicks on two channels into a histogram, where all ticks are considered both as "start" and "stop" clicks and both positive and negative time differences are considered.

class Correlation(*tagger*, *channel_1*, *channel_2*, *binwidth*, *n_bins*)

Parameters

- **tagger** (TimeTagger) – time tagger object
- **channel_1** (*int32*) – channel on which (stop) clicks are received
- **channel_2** (*int32*) – channel on which reference clicks (start) are received (when left empty or set to *CHANNEL_UNUSED* -> an auto-correlation measurement is performed, which is the same as setting *channel_1* = *channel_2*)

- **binwidth** (*uint64*) – bin width in ps
- **n_bins** (*uint*) – the number of bins in the resulting histogram

getData()
Returns a one-dimensional array of size *n_bins* containing the histogram.

getDataNormalized()
Return the data normalized as:

$$g^{(2)}(\tau) = \frac{\Delta t}{binwidth \cdot N_1 \cdot N_2} \cdot histogram(\tau)$$

where Δt is the capture duration, N_1 and N_2 are number of events in each channel.

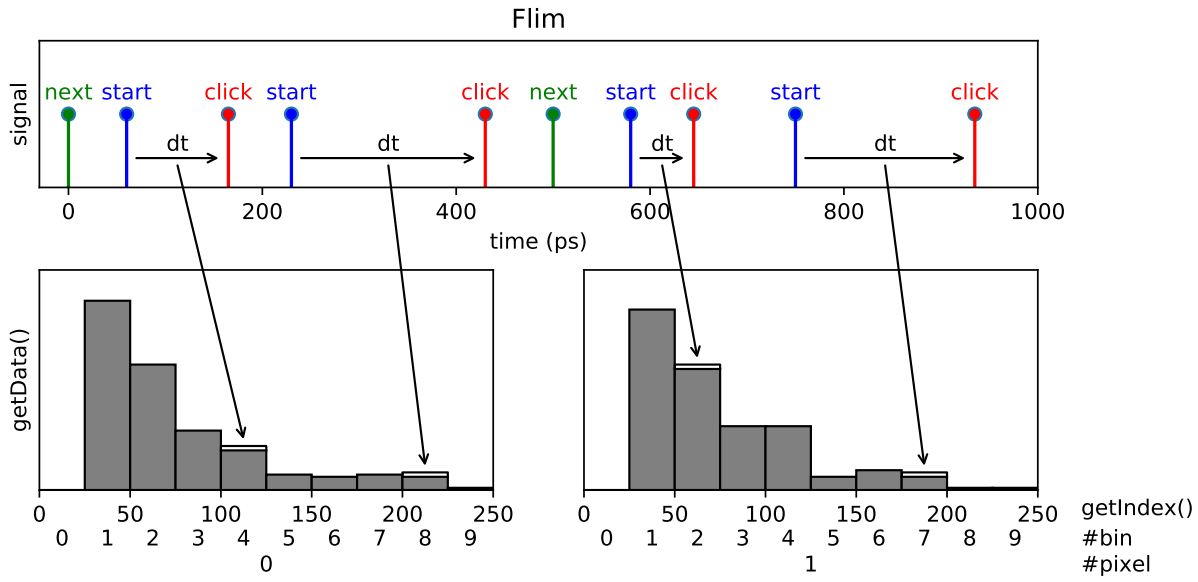
getIndex()
Returns a vector of size *n_bins* containing the time bins in ps.

clear()
Resets the accumulated data.

7.7.5 Advanced time histograms

This section describes advanced time histogramming measurements that simplify building more complex measurements and various imaging techniques.

FLIM



Fluorescence-lifetime imaging microscopy (FLIM) is an imaging technique for producing an image based on the differences in the exponential decay rate of the fluorescence from a sample.

Fluorescence lifetimes can be determined in the time domain by using a pulsed source. When a population of fluorophores is excited by an ultrashort or delta-peak pulse of light, the time-resolved fluorescence will decay exponentially.

This measurement implements a line scan in a FLIM (Fluorescence-lifetime imaging microscopy) image that consists of a sequence of pixels. This could either represent a single line of the image, or - if the image is represented as a single meandering line - this could represent the entire image.

This measurement is a special case of the more general *TimeDifferences* measurement.

The measurement successively acquires n histograms (one for each pixel in the line scan), where each histogram is determined by the number of bins and the bin width.

class Flim(*tagger*, *click_channel*, *start_channel*, *next_channel*, *binwidth*, *n_bins*, *n_pixels*)

Parameters

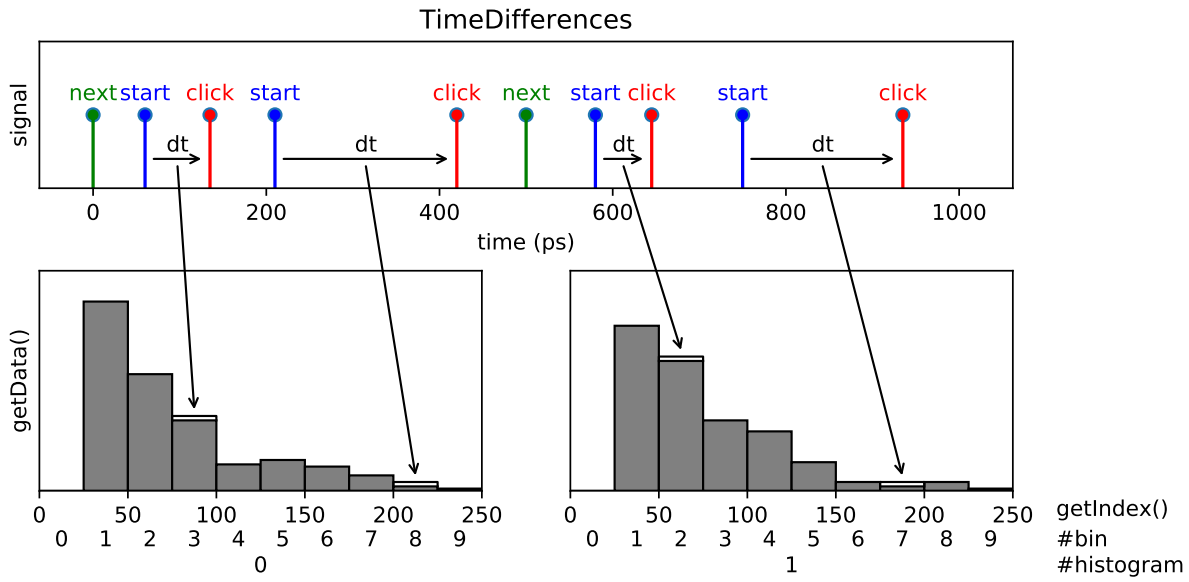
- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which clicks are received
- **start_channel** (*int32*) – channel on which start clicks are received
- **next_channel** (*int32*) – channel on which pixel triggers are received
- **binwidth** (*int64*) – bin width in ps
- **n_bins** (*int32*) – number of bins in each histogram
- **n_pixels** (*int32*) – number of pixels

getData()
Returns a two-dimensional array of size *n_bins* by *n_pixels* containing the histograms.

getIndex()
Returns a vector of size *n_bins* containing the time bins in ps.

clear()
Resets the array to zero.

TimeDifferences



A multidimensional histogram measurement with the option up to include three additional channels that control how to step through the indices of the histogram array. This is a very powerful and generic measurement. You can use

it to record cross-correlation, lifetime measurements, fluorescence lifetime imaging and many more measurements based on pulsed excitation. Specifically, the measurement waits for a tag on the *start_channel*, then measures the time difference between the start tag and all subsequent tags on the *click_channel* and stores them in a histogram. If no *start_channel* is specified, the *click_channel* is used as *start_channel* corresponding to an auto-correlation measurement. The histogram has a number *n_bins* of bins of bin width *binwidth*. Clicks that fall outside the histogram range are discarded. Data accumulation is performed independently for all start tags. This type of measurement is frequently referred to as 'multiple start, multiple stop' measurement and corresponds to a full auto- or cross-correlation measurement.

The data obtained from subsequent start tags can be accumulated into the same histogram (one-dimensional measurement) or into different histograms (two-dimensional measurement). In this way, you can perform more general two-dimensional time-difference measurements. The parameter *n_histograms* specifies the number of histograms. After each tag on the *next_channel*, the histogram index is incremented by one and reset to zero after reaching the last valid index. The measurement starts with the first tag on the *next_channel*.

You can also provide a synchronization trigger that resets the histogram index by specifying a *sync_channel*. The measurement starts when a tag on the *sync_channel* arrives with a subsequent tag on *next_channel*. When a rollover occurs, the accumulation is stopped until the next sync and subsequent next signal. A sync signal before a rollover will stop the accumulation, reset the histogram index and a subsequent signal on the *next_channel* starts the accumulation again.

Typically, you will run the measurement indefinitely until stopped by the user. However, it is also possible to specify the maximum number of rollovers of the histogram index. In this case, the measurement stops when the number of rollovers has reached the specified value. This means that for both a one-dimensional and for a two-dimensional measurement, it will measure until the measurement went through the specified number of rollovers / sync tags.

class TimeDifferences(*tagger*, *click_channel*, *start_channel*, *next_channel*, *sync_channel*, *binwidth*, *n_bins*, *n_histograms*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which stop clicks are received
- **start_channel** (*int32*) – channel that sets start times relative to which clicks on the click channel are measured
- **next_channel** (*int32*) – channel that increments the histogram index
- **sync_channel** (*int32*) – channel that resets the histogram index to zero
- **binwidth** (*int64*) – binwidth in picoseconds
- **n_bins** (*int32*) – number of bins in each histogram
- **n_histograms** (*int32*) – number of histograms

getData()
Returns a two-dimensional array of size *n_bins* by *n_histograms* containing the histograms.

getIndex()
Returns a vector of size *n_bins* containing the time bins in ps.

clear()
Resets all data to zero.

setMaxCounts()
Sets the number of rollovers at which the measurement stops integrating. To integrate infinitly, set the value to 0, which is the default value.

getCounts()
Returns the number of rollovers (histogram index resets).

ready()
Returns 'true' when the required number of rollovers set by *setMaxCounts()* has been reached.

Overflow handling

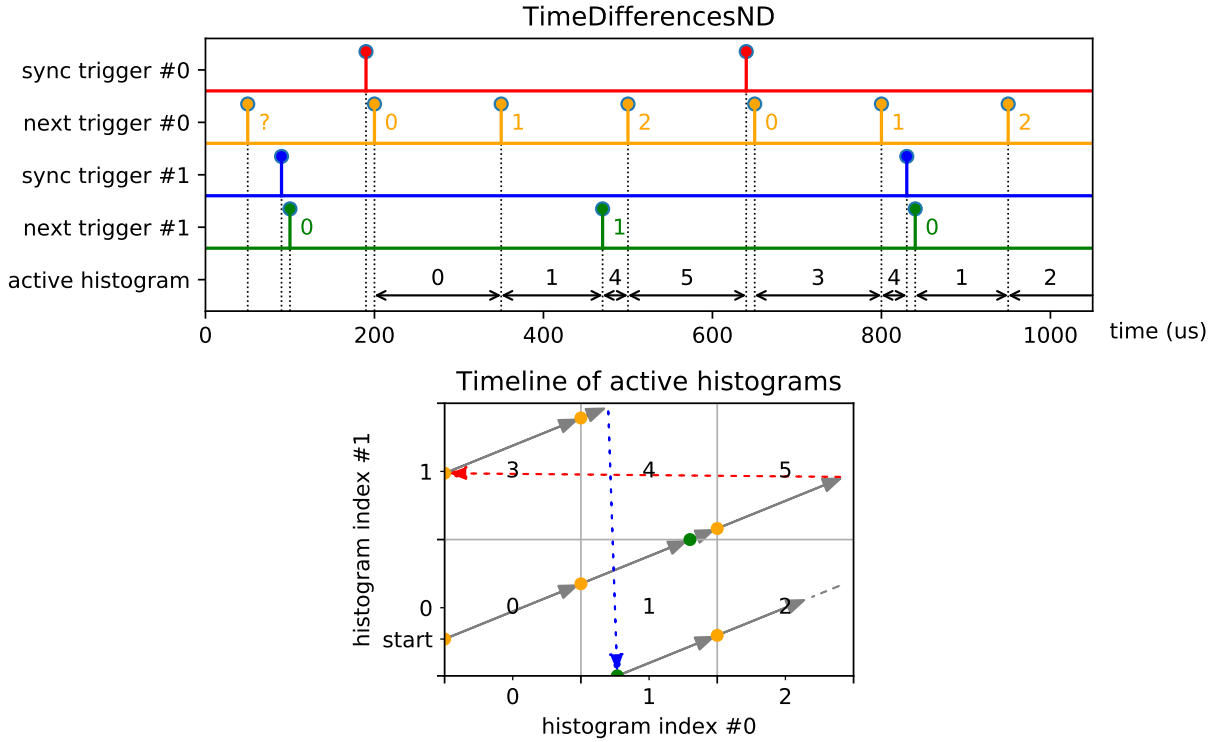
The different ways overflows are handled depend on whether a next_channel and a sync_channel is defined:

sync_channel and next_channel are both defined the measurement stops integrating at an overflow and continues with the next signal on the sync_channel

only next_channel is defined the histogram index is reset at the overflow and the next signal on the next_channel starts the integration again

sync_channel and next_channel are both undefined the accumulation continues

TimeDifferencesND



Accumulates the time differences between clicks on two channels in a multi-dimensional histogram.

This is a multidimensional implementation of the *TimeDifferences* measurement class. Please read their documentation first.

This measurement class extends the *TimeDifferences* interface for a multidimensional amount of histograms. It captures many multiple start - multiple stop histograms, but with many asynchronous *next_channel* triggers. After each tag on each *next_channel*, the histogram index of the associated dimension is incremented by one and reset to zero after reaching the last valid index. The elements of the parameter *n_histograms* specify the number of histograms per dimension. The accumulation starts when *next_channel* has been triggered on all dimensions.

You should provide a synchronization trigger by specifying a sync_channel per dimension. It will stop the accumulation when an associated histogram index rollover occurs. A sync event will also stop the accumulation, reset the

histogram index of the associated dimension, and a subsequent event on the corresponding *next_channel* starts the accumulation again. The synchronization is done asynchronous, so an event on the *next_channel* increases the histogram index even if the accumulation is stopped. The accumulation starts when a tag on the *sync_channel* arrives with a subsequent tag on *next_channel* for all dimensions.

Please use `TimeTagger.setInputDelay()` to adjust the latency of all channels. In general, the order of the provided triggers including maximum jitter should be:

old start trigger –> all sync triggers –> all next triggers –> new start trigger

class TimeDifferencesND(*tagger*, *click_channel*, *start_channel*, *next_channels*, *sync_channels*, *n_histograms*, *binwidth*, *n_bins*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **click_channel** (*int32*) – channel on which stop clicks are received
- **start_channel** (*int32*) – channel that sets start times relative to which clicks on the click channel are measured
- **next channels** (*list[int32]*) – vector of channels that increments the histogram index
- **sync channels** (*list[int32]*) – vector of channels that resets the histogram index to zero
- **n_histograms** (*int32*) – vector of numbers of histograms per dimension
- **binwidth** (*int64*) – width of one histogram bin in ps
- **n_bins** (*int32*) – number of bins in each histogram

7.7.6 Timetag streaming

Measurement classes described in this section provide direct access to the time tag stream with minimal or no preprocessing.

Time tag format

The time tag contain essential information about the detected event and have the following format:

Size	Type	Description
8 bit	enum *OverflowType*	overflow type
8 bit	–	reserved
16 bit	uint16	number of missed events
32 bit	int32	channel number
64 bit	int64	time in ps from device start-up

enum *OverflowType*

This enumeration describes the overflow condition.

TimeTag = 0 - a normal event from any input channel, no overflow.
Error = 1 - an error in the internal data processing, e.g. on plugging the external clock. This invalidates the global time.
OverflowBegin = 2 - marks the beginning of an interval with incomplete data because of too high data rates.
OverflowEnd = 3 - marks the end of the interval. All events, which were lost in this interval, have been handled
MissedEvents = 4 - this virtual event signals the amount of lost events per channel within an overflow interval. Might be sent repeatedly for larger amounts of lost events.

TimeTagStream

Access the time tag stream. A buffer of the size *max_tags* is filled with the incoming time tags. As soon as *getData()* is called the current buffer is returned and incoming tags are stored in a new, empty buffer.

class TimeTagStream(*tagger*, *max_tags*, *channels*)

Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **max_tags** (*int32*) – buffer size for storing time tags
- **channels** (*list[int32]*) – which are dumped to the file (when empty or not passed all active channels are dumped)

getData()
Returns a *TimeTagStreamBuffer* object and clears the internal buffer of the *TimeTagStream* measurement. Clearing the internal buffer on each call to *getData()* guarantees that consecutive calls to this method will return every timetag only once, also with no data loss. The returned *TimeTagStreamBuffer* object contains a vector for the channels, the timestamps in ps and the overflow flags, of the timetags.

class TimeTagStreamBuffer

getTimestamps()
Returns an array of timestamps.

Returns Event timestamps in picoseconds for all chosen channels.

Return type list(int64)

getChannels(*self*)
Returns an array of channel numbers for every timestamp.

Returns Channel number for each detected event.

Return type list(int64)

getOverflows(*self*)
Deprecated since version 2.5.0.

Returns an array of overflow flags for every timestamp.

getEventTypes(*self*)
Returns an array of event type for every timestamp. See, *Time tag format*.

Returns Event type value for each detected event.

Return type list(OverflowType)

getMissedEvents()
Returns an array of missed event counts.

hasOverflows()
Returns True if overflow was detected in any of the tags received.

Returns True/False

FileWriter

Writes the timetag stream into a file in a binary format with a lossless compression. The estimated file size requirements are 2-4 bytes per event under typical conditions. The files produced by this measurement are significantly smaller than those created with the *Dump* class. The files created with *FileWriter* measurement can be read using *FileReader* or loaded into the Virtual Time Tagger.

The *FileWriter* is able to split the data into a separate file seamlessly when the file size reaches a maximal size. For the file splitting to work properly, the filename specified by the user will be extended with a suffix containing sequential counter, so the filenames will look like in the following example

```
fw = FileWriter(tagger, 'filename.ttbin', [1,2,3]) # Store tags from channels 1,2,3

# When splitting occurs the files with following names will be created
#    filename.ttbin     # the sequence header file with no data blocks
#    filename.1.ttbin   # the first file with data blocks
#    filename.2.ttbin
#    filename.3.ttbin
#    ...
```

In addition, the *FileWriter* will query and store the configuration of the Time Tagger in the same format as returned by the *TimeTagger.getConfiguration()* method. The configuration is always written into every file.

class FileWriter(*tagger*, *filename*, *channels*)

Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **filename** (*string*) – name of the file to store to
- **channels** (*list[int32]*) – non-empty list of real or virtual channels

Class constructor. As with all other measurements, the data recording starts immediately after the class instantiation.

Note: Compared to the *Dump* measurement, the *FileWriter* requires explicit specification of the channels. If you want to store timetags from all input channels, you can query the list of all input channels with *TimeTagger.getChannelList()*.

---

split([*new_filename*=""])
Close the current file and create a new one. If the *new_filename* is provided, the data writing will continue into the file with the new filename and the sequence counter will be reset to zero.

You can force the file splitting when you call this method without parameter or when the *new_filename* is an empty string.

Parameters **new_filename** (*string*) – filename of the new file.

setMaxFileSize(*max_file_size*)
Set the maximum file size on disk. When this size is exceeded a new file will be automatically created to continue recording. The actual file size might be larger by one block. (default: ~1 GByte)

getMaxFileSize()
Returns the maximal file size. See also *FileWriter.setMaxFileSize()*.

getTotalEvents()
Returns the total number of events written into the file(s).

getTotalSize()
Returns the total number of bytes written into the file(s).

FileReader

This measurement allows you to read data files store with *FileReader*. The *FileReader* reads a data block of the specified size into a *TimeTagStreamBuffer* object and returns this object. The returned data object is exactly the same as returned by the *TimeTagStream* measurement and allows you to create a custom data processing algorithms that will work both, for reading from a file and for the on-the-fly processing.

The *FileReader* will automatically recognize if the files were split and read them too one by one.

Example:

```
# Lets assume we have following files created with the FileWriter
#  measurement.ttbin     # sequence header file with no data blocks
#  measurement.1.ttbin   # the first file with data blocks
#  measurement.2.ttbin
#  measurement.3.ttbin
#  measurement.4.ttbin
#  another_meas.ttbin
#  another_meas.1.ttbin

# Read all files in the sequence 'measurement'
fr = FileReader("measurement.ttbin")

# Read only the first data file
fr = FileReader("measurement.1.ttbin")

# Read only the first two files
fr = FileReader(["measurement.1.ttbin", "measurement.2.ttbin"])

# Read the sequence 'measurement' and then the sequence 'another_meas'
fr = FileReader(["measurement.ttbin", "another_meas.ttbin"])
```

class FileReader(*filenames*)
This is the class contructor. The *FileReader* automatically continues to read files that were split by the *FileWriter*.

Parameters **filenames** (*list[string]*) – filename(s) of the files to read.

getData(*size_t n_events*)

Parameters **n_events** (*int*) – Number of timetags to read fromt the file.

Reads the next *n_events* and returns the buffer object with the specified number of timetags. The FileReader stores the current location in the data file and guarantees that every timetag is returned exacltly once. If less than *n_elements* are returned, the reader has reached the end of the last file in the file-list *filenames*. To check if more data is available for reading, it is more convenient to use *hasData()*.

hasData()
Returns *True* if more data is available for reading. Returns *False* if all data has been read from all the files specified in the class contructor.

getConfiguration()
Returns a JSON formatted string that contains the Time Tagger configuration at the time of file creation.

Dump

Deprecated - please use *FileWriter* instead.

Writes the timetag stream into a file in a binary format. See also, *Time tag format*.

Please visit the programming examples provided in the installation folder of how to dump and load data.

class Dump(*tagger*, *filename*, *max_tags*, *channels*)

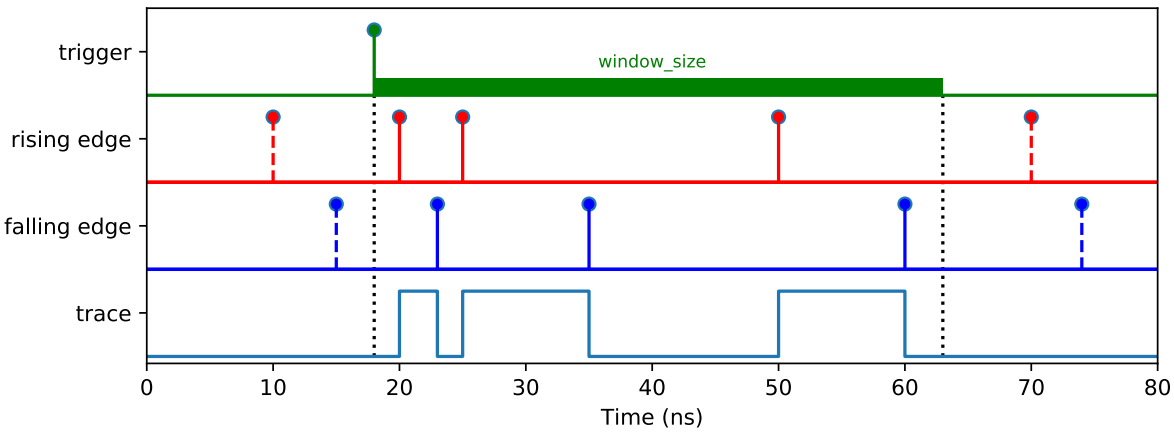
Parameters

- **tagger** (TimeTagger) – time tagger object instance
- **filename** (*string*) – name of the file to dump to
- **max_tags** (*int64*) – stop after this number of tags has been dumped. Negative values will dump forever
- **channels** (*list[int32]*) – list of real or virtual channels which are dumped to the file (when empty or not passed all active channels are dumped)

clear()
Delete current data in the file and restart data storage.

stop()
Stops data recording and closes data file.

Scope



The *Scope* class allows to visualize time tags for rising and falling edges in a time trace diagram similarly to an ultrafast logic analyzer. The trace recording is synchronized to a trigger signal which can be any physical or virtual channel. However, only physical channels can be specified to the *event_channels* parameter. Additionally, one has to specify the time *window_size* which is the timetrace duration to be recorded, the number of traces to be recorded and

the maximum number of events to be detected. If `n_traces < 1` then retriggering will occur infinitely, which is similar to the "normal" mode of an oscilloscope.

Note: Scope class implicitly enables the detection of positive and negative edges for every physical channel specified in *event_channels*. This accordingly doubles the data rate requirement per input.

---

class Scope(*tagger*, *event_channels=[]*, *trigger_channel*, *window_size*, *n_traces*, *n_max_events*)

Parameters

- **tagger** (TimeTagger) – TimeTagger object
- **event_channels** (*list[int32]*) – List of channels
- **trigger_channel** (*int32*) – Channel number of the trigger signal
- **window_size** (*int64*) – Time window in picoseconds
- **n_traces** (*int32*) – Number of trigger events to be detected
- **n_max_events** (*int32*) – Max number of events to be detected

getData()

Returns an array of the size equal to the number of *event_channels*, where each element is an array of event structures with fields {state, time}.

Data can be extracted as shown in the pseudo-python code below.

```
for channel in scope.getData():
    for event in channel:
        t.append(event.time)
        val.append(event.value)
    plot_steps(t, val)  # for example "matplotlib.pyplot.step"
```

Returns Array of event arrays for each channel.

Return type Event[][struct{state, time}]

7.7.7 Helper classes

SynchronizedMeasurements

The *SynchronizedMeasurements* class allows for synchronizing multiple measurement classes in a way that ensures all these measurements to start, stop simultaneously and operate on exactly the same time tags. To disable the autostart of the measurements for *SynchronizedMeasurements*, a proxy-object from a SynchronizedMeasurements object (*synchronizedMeasurements.getTagger()*) can be passed to the measurements at initialization avoiding the autostart.

class SynchronizedMeasurements(*tagger*)

Parameters **tagger** (TimeTagger) – TimeTagger object

registerMeasurement(*measurement*)

Registers the *measurement* object into a pool of the synchronized measurements.

Note: Registration of the measurement classes with this method does not synchronize them. In order to start/stop/clear these measurements synchronously, call these functions on the

SynchronizedMeasurements object after registering the measurement objects, which should be synchronized.

Note: Using *registerMeasurement()*, it is not possible to synchronize the *Histogram* and *Flim* measurements. It is recommended to use *getTagger()* to synchronize measurements. If you have to use *registerMeasurement()*, please take *TimeDifferences* as a replacement as shown below.

Equivalency between `Histogram` and `TimeDifferences`:

```
Histogram(tagger, click_channel=1, start_channel=2,
    binwidth=100, n_bins=1000)

# Histogram using TimeDifferences
TimeDifferences(tagger, click_channel=1, start_channel=2,
    next_channel=CHANNEL_UNUSED, sync_channel=CHANNEL_UNUSED,
    binwidth=100, n_bins=1000, n_histograms=1)
```

Equivalency between `Flim` and `TimeDifferences`:

```
Flim(tagger, click_channel=1, start_channel=2, next_channel=3,
    binwidth=100, n_bins=1000, n_pixels=320*240)

# FLIM using TimeDifferences
TimeDifferences(tagger, click_channel=1, start_channel=2,
    next_channel=3, sync_channel=CHANNEL_UNUSED,
    binwidth=100, n_bins=1000, n_histograms=320*240)
```

Parameters **`measurement`** – Any measurement (IteratorBase) object.

`unregisterMeasurement`(*measurement*)
Unregisters the *measurement* object out of the pool of the synchronized measurements.

Note: This method does nothing if the provided measurement is not currently registered.

Parameters **`measurement`** – Any measurement (IteratorBase) object.

`start`()
Calls *start()* for every registered measurement in a synchronized way.

`startFor`(*duration*[, *clear=True*])
Calls *startFor()* for every registered measurement in a synchronized way.

`stop`()
Calls *stop()* for every registered measurement in a synchronized way.

`clear`()
Calls *clear()* for every registered measurement in a synchronized way.

`isRunning`()
Calls *isRunning()* for every registered measurement and returns true if any measurement is running.

getTagger()

Returns a proxy tagger object which can be passed to the constructor of a measurement class to register the measurements at initialization to the synchronized measurement object. Those measurements will not start automatically.

Note: The proxy tagger object returned by *getTagger()* is not identical with the *TimeTagger* object created by `TimeTagger.createTimeTagger()`. You can create synchronized measurements with the proxy object the following way:

```
tagger = TimeTagger.createTimeTagger()
syncMeas = TimeTagger.SynchronizedMeasurements(tagger)
taggerSync = syncMeas.getTagger()
counter = TimeTagger.Counter(taggerSync, [1, 2])
countrate = TimeTagger.Countrate(taggerSync, [3, 4])
```

Passing *tagger* as a constructor parameter would lead to the not syncrhonized behavior.

CHAPTER

EIGHT

IN DEPTH GUIDES

8.1 Conditional Filter

The Conditional Filter is a hardware feature that allows you to remove irrelevant time tags carrying no information. In a typical use case, you have a high-frequency signal applied to at least one channel. Examples include fluorescence lifetime measurements or optical quantum information and cryptography where you want to capture synchronization clicks from a high repetition rate excitation laser.

The Conditional Filter distinguishes between *trigger* channels and *filtered* channels. All input channels of your Time Tagger are fully equivalent and can be used as both, trigger or filtered channels. The data rate of the filtered channels will be reduced. The reduction is controlled by the trigger channels: Every trigger opens the gate for exactly one event per filtered channel. All other events in the filtered channels will be discarded on the Time Tagger and do not need to be transferred via the USB connection.

Being a hardware feature, the Conditional Filter is not controlled on the level of individual measurements. It is enabled on the level of your physical device with a typical Python code looking like

```
import TimeTagger
tagger = TimeTagger.createTimeTagger()
tagger.setConditionalFilter(trigger=[1], filtered=[8])
```

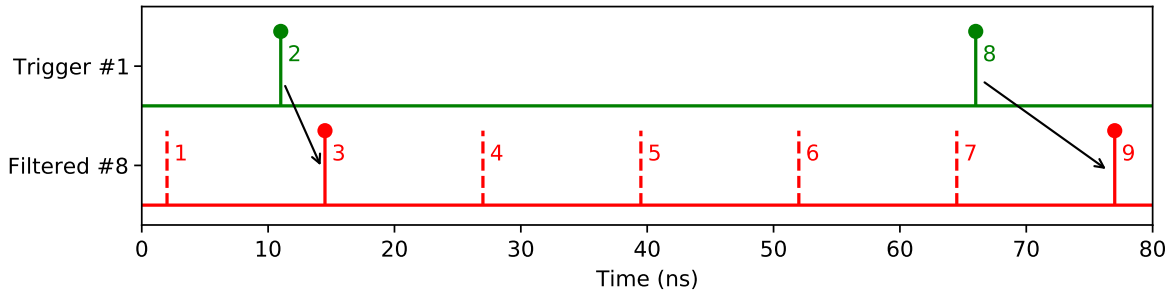
The details will be explained in the *Setup of the Conditional Filter* section.

8.1.1 Example configurations

One trigger and one filtered channel

The most fundamental case involves one filtered-channel and one trigger-channel:

```
tagger.setConditionalFilter(trigger=[1], filtered=[8])
```

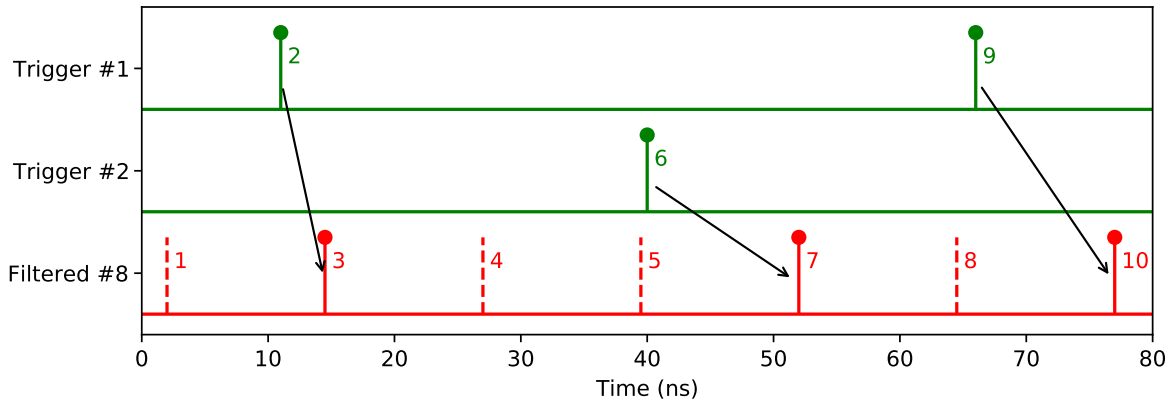


The Conditional Filter discards by default all signals of the filtered-channel. Only the very next event is transmitted after an event on the trigger-channel. In the example, click 2 opens the gate for click 3. When click 3 passes, it closes the gate and the subsequent events will be discarded until another event (click 8) occurs in the trigger channel.

Multiple trigger-channels

There is the option to define more than one trigger-channel for the Conditional Filter. As a consequence, the next event on the filtered-channel is transmitted when there was a event at *any* of the trigger-channels:

```
tagger.setConditionalFilter(trigger=[1, 2], filtered=[8])
```

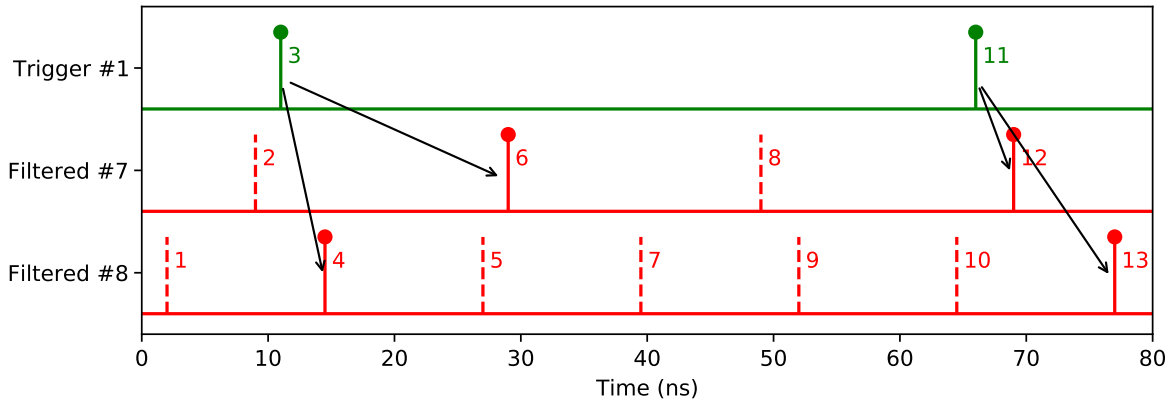


This is the typical use case when you detect photons with multiple detectors and want to correlate both with the common excitation laser.

Multiple filtered channels

It is also possible to use the Conditional Filter with one trigger-channel and several filtered-channels:

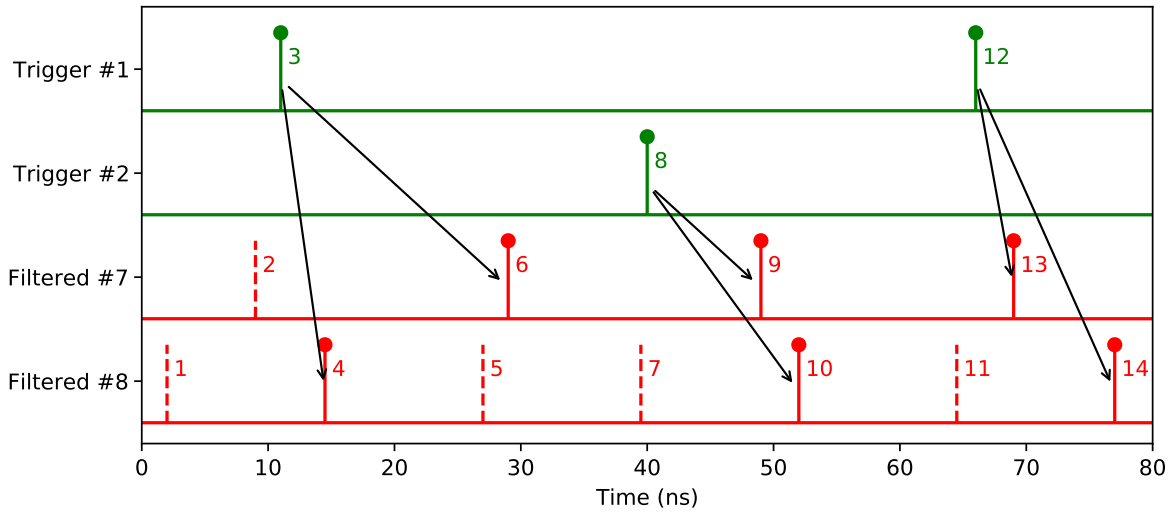
```
tagger.setConditionalFilter(trigger=[1], filtered=[7, 8])
```



Multiple trigger and filtered channels

In general, you can also combine multiple trigger-channels and multiple filtered-channels:

```
tagger.setConditionalFilter(trigger=[1, 2], filtered=[7, 8])
```



This scheme shows two different high-frequency signals on channels #7 and #8. Such cases can occur when you want to run two completely independent experiments on a single Time Tagger. For instance, channels #1/#7 and #2/#8 may represent the two experiments. It is not possible to set up two independent Conditional Filters for these groups. The scheme shown is the only way to apply the Conditional Filter in this case - with the drawback that channel #1 (#2) may also trigger channel #8 (#7), making the filtering less efficient.

8.1.2 Understanding the filtering mechanism

The Conditional Filter is a hardware feature that is embedded in a sequence of processing stages. It is important to understand the order of these stages. Some unexpected results can occur when you are not aware of these mechanisms, so read the following section with care.

Terms

Input time stamp This is the time stamp *you* are interested in: It refers to the time when the input signal transits the trigger level at the input connector.

TDC time stamp This is the time stamp *the Time Tagger* is interested in: It is the raw 64 bit integer the FPGA attributes to a pulse edge.

Hardware delay The signal entering the input connector is routed through the Time Tagger into the FPGA where the time to digital conversion is performed. This route differs from channel to channel and so does the accumulated delay. Because of this, we need to distinguish between *Input time stamp* and *TDC time stamp*. The *hardware delay* cannot be controlled by the user, it is defined by the design of the Time Tagger hardware and the FPGA configuration (this can vary from software release to software release). But don't worry, the Time Tagger software is calibrated to compensate for this delay. Except for the purpose of understanding the Conditional Filter, you do not need to care about it.

External delay Any delay introduced before the Time Tagger, e.g. by cable lengths or optical pathways.

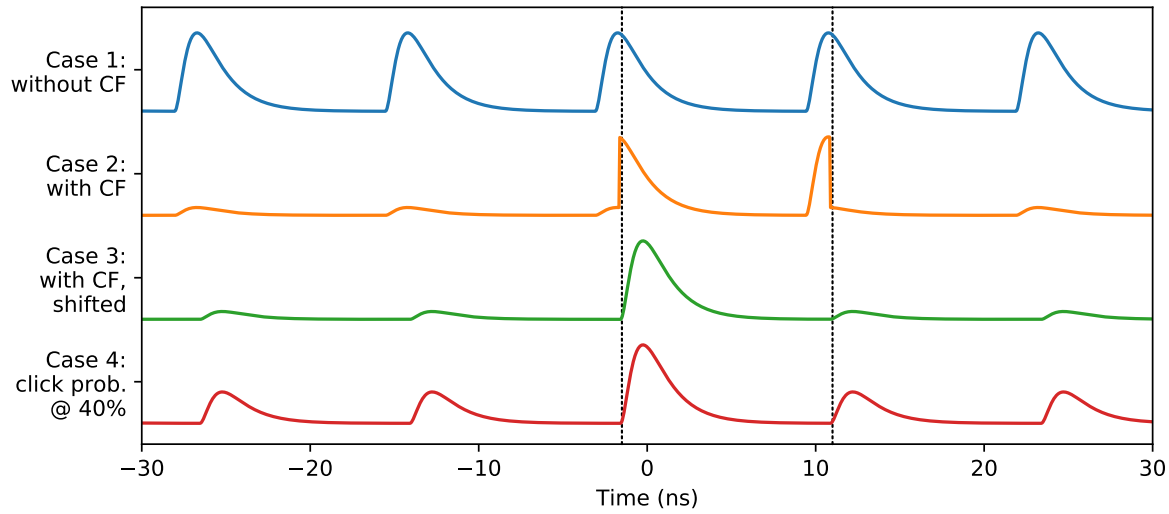
Processing stages

1. **Pulse enters the Time Tagger:** Up to the input connector, the user is in charge of the *external delays*. They can be controlled by changing cable lengths or optical pathways. The time tag generated by the Time Tagger should therefore represent the temporal order at the input connectors. This is the *input time stamp*.
2. **Time to digital conversion:** The pulses propagate through the Time Tagger. They are compared to the trigger level of the input stage. This results in a high or low logic level. This is still analog information that propagates to the FPGA. Here, the 64bit integer value (the *TDC time stamp*) is attributed to the pulse edge. The propagation length up to this time to digital conversion (TDC) differs from channel to channel. This needs to be kept in mind.
3. **Conditional Filter:** As a first filter stage, the Conditional Filter is applied. The time tags of trigger channels and filtered channels are compared. This happens based on the raw *TDC time stamp*. The time order of these stamps can deviate from the order of the *input time stamps* that you are dealing with usually.
4. **Event Divider:** As a second filter stage, the Event Divider can be applied. Only every n-th time tag is transmitted, all others are dismissed.
5. **The bottleneck - USB transfer:** The time tags are buffered and transmitted to the PC. At this point, after applying Conditional Filter and Event Divider, it is important that the resulting data rate on average does not exceed the maximum data rate.
6. **Hardware delay compensation/setInputDelay:** From now on, the Time Tagger hardware is not involved anymore. Now the Time Tagger software compensates the *TDC time stamp* for the *hardware delay* to provide you the *input time stamp* (it is possible to disable the hardware delay compensation, see *Control hardware delay compensation*). Additionally, you can modify this compensation by `setInputDelay()`.
7. **Delayed Channel:** The most flexible way to control the relative delay of your signals are Virtual Channels.

Consequences

The nature of the filtering process can produce unintuitive results that need to be handled. We will explore these cases based on the example of a fluorescence lifetime measurement. The sample is excited by a pulsed laser with a repetition rate of 80 MHz (period of 12.5 ns), the laser synchronization signal is connected to channel #8. So channel #8 is the high-frequency input that needs to be filtered. Fluorescence photons are collected by a single-photon detector connected to channel #1 that will trigger the Conditional Filter. We set up a correlation measurement and look at different cases:

```
TimeTagger.Correlation(tagger, 1, 8)
```

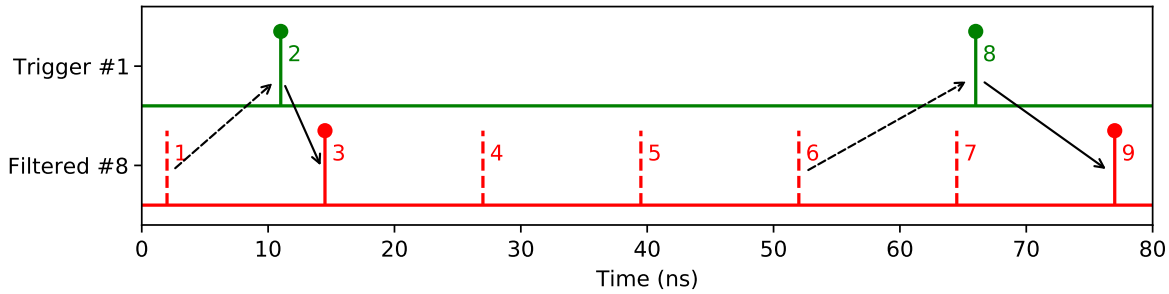


Case 1: Without the Conditional Filter set up, the Correlation measurement class provides a periodic signal. The periodicity is a result of the multi-start/multi-stop approach of the Correlation measurement: A click on the detector will contribute together with any laser synchronization pulse to the correlation, not only with the one that actually stimulated the photon. Without the Conditional Filter, there will be a laser time tag every 12.5 ns. Because this high frequency cannot be transferred for a long time, buffer overflows will lead to discarded data.

Case 2: With the Conditional Filter on, the data rate is highly reduced at the cost of losing the full periodicity of the signal:

```
tagger.setConditionalFilter(trigger=[1], filtered=[8])
```

Now we observe that the majority of the events is in the range of a few nanoseconds. However, the signal does not look like expected: Instead of a signal resembling one of the peaks from Case 1, a double peak appears. If you look carefully at the signal, you can see that the lifetime curve is cut along the dotted line and one part is shifted by one period. This indicates that the physical delay between the input channels is not designed properly. The scheme illustrates the problem:



The dashed line indicates which pulse excited the sample. If the photon is emitted early by the sample (click 2), it will trigger the first pulse (click 3) after the stimulating one (click 1). In the second case, the photon is emitted late and the subsequent laser pulse (click 7) has already passed. In this case, click 9 is passed and click 8 seems to be very early, although it is quite late, in fact.

Case 3: To align the signal properly, having the signal in between two laser events, you need to adjust your *external delays*. You might either modify optical path lengths or use cables of different lengths. If you look through the *Processing stages*, there is no other way to manipulate the delay inside the Time Tagger *before* the Conditional Filter. `setInputDelay()`, for instance, is applied after the Conditional Filter and will have no effect on the selection of the filtered pulse.

Case 4: This case illustrates that the height of the higher-order peaks is determined by the count rate of your detector. The relative height (compared to the center peak) is proportional to the probability for a laser synchronization pulse to pass the Conditional Filter in the higher-order period. This probability is given by the probability that a detector click occurs in the respective period and gates the synchronization click. In Case 1, without the Conditional Filter, the probability is 100% - every synchronization pulse is passed. For Case 2 and Case 3, the probability has been set to 10%, in Case 4 it has been increased to 40%.

Note: In Cases 3 and 4, with *external delays* well adjusted to each other, you can see a signal at negative times. How is this possible? Wouldn't this mean that the laser synchronization click arrived earlier than the photon click that gated it? Does my Time Tagger violate causality?

The answer is: No, it does not. The occurrence of negative delays is caused by the difference between the *input time stamps* and the *TDC time stamps*. Negative delays occur in *input time stamps*, but causality must only be obeyed in *TDC time stamps*. The occurrence of negative delays indicates that the *hardware delay* of channel #8 (laser synchronization) is larger than that of channel #1 (detector).

8.1.3 Setup of the Conditional Filter

The `setConditionalFilter()` method expects two arguments, *trigger* and *filtered*, and accepts the optional boolean argument *hardwareDelayCompensation*:

```
tagger.setConditionalFilter(trigger: list[int32],
                            filtered: list[int32],
                            hardwareDelayCompensation: bool = True)
```

The effect of *trigger* and *filter* can be reviewed in the *Example configurations* section.

Control hardware delay compensation

With the argument *hardwareDelayCompensation* you can decide whether the *hardware delay* is compensated or not. This means, in fact, that you can decide whether you work with *input time stamps* or with *TDC time stamps*.

hardwareDelayCompensation = True (default)

Pros

- Time tags are provided in the way you are used to it
- The signal position will not depend on the software version

Cons

- Negative time differences can occur between trigger-channel and filtered-channel and seemingly violate causality

hardwareDelayCompensation = False

Pros

- Provided Time tags will be in the same temporal order as for the ConditionalFilter, no negative time differences will occur

Cons

- Signal positions may change upon software update
- Affects all channels, not only the ones listed in *trigger* and *filtered*.

Disable the Conditional Filter

To disable the Conditional Filter, you can either pass empty lists or use the `clearConditionalFilter()` method:

```
tagger.setConditionalFilter([], [])
# or
tagger.clearConditionalFilter()
```

8.2 Synchronization of the Time Tagger pipeline

In order to achieve a real-time evaluation of the events with high data rates, the Time Tagger series uses a pipeline based parallel processing.

The hardware records a timestamp for every incoming event and stores it in a large on-device buffer. The size of this buffer can be configured with `setHardwareBufferSize()`. The buffer contents are read by computer over USB, typically in blocks of 128k events or when the time between the blocks exceeds 20 ms. Waiting until a block of data is available is aimed at optimizing the USB throughput while limiting the time between consecutive block allows for reducing data latency on slow event rates. The block size can be tuned by a user with `setStreamBlockSize()`. On the computer, the blocks of data are processed by all running measurements in the order in which the measurements were created. Only one measurement has access to a block at any given time. Once a measurement has finished processing the block, it is ready to process the next block while the previous block becomes available to the next measurement.

Naturally, the transferring and processing of the data takes time and results in the latency. The latency between signal arrival and its appearance in the measurement data is usually below 100 ms; however, it can become as large as a few seconds if the on-device buffer fills up faster than the computer can transfer and process the data.

Proper operation of the pipeline and the control of the device parameters requires a suitable synchronization method. Time Tagger uses the concept of fencing. A fence is a unique identifier that is sent by the software to the hardware. It is added at the end of the on-device buffer data, streamed back to the computer along with timestamp data, and processed by all measurement classes. Once the Time Tagger software detects the fence, it knows that it is located at the data position which was in the buffer when the fence was created. The usefulness of fencing is easily demonstrated with a following example. When you create a measurement, you expect that it starts processing data from that very instance of time; however, it starts processing the data, which was recorded earlier and is already available in the buffer. With fencing, the measurement creates a fence and begins data accumulation only when it receives the fence back. In this way, the measurement is dealing with the data recorded as close to the measurement creation as possible and avoids processing of the older data.

You can use the fencing mechanism manually. First, you have to create a new fence with `getFence()` and then wait for it to be signaled with `waitForFence()` at any time later. If you want to create a fence and immediately wait for it then using the `sync()` method is more convenient.

CHAPTER

NINE

LINUX

The package installs the Python and C++ libraries for amd64 systems including example programs.

Graphical user interface (web application):

- Launch via *timetagger* from the console or from the application launcher.

Known issues

- In case you have installed a previous version of the Time Tagger software, please reset the cache of your browser.
- Closing the web application server can cause an error message to appear.

Using the Time Tagger with Python 2.7 or 3:

- Install *numpy* (e.g. *pip install numpy*), which is required for the Time Tagger libraries.
- The Python libraries are installed in your default Python search path: */usr/lib/pythonX.Y/dist-packages/* or */usr/lib64/pythonX.Y/site-packages/*.
- The examples can be found within the */usr/lib/timetagger/examples/python/* folder.

Using the Time Tagger with C++:

- The examples can be found within the */usr/lib/timetagger/examples/cpp/* folder.
- The header files can be found within the */usr/include/timetagger/* folder (-I /usr/include/timetagger).
- The assembly shall be linked with */usr/lib/libTimeTagger.so* (-l TimeTagger).

General remark:

- Please contact us in case you have any questions or comments about the Ubuntu or CentOS package and/or the API for the Time Tagger.
- The official supported linux distributions are Ubuntu 16.04, 18.04 and 20.04, and CentOS 7 and 8.

  However, the C++ interface will likely also work on other distributions out of the box. The source of the Python wrapper *_TimeTagger.cxx* is provided in */usr/lib64/pythonX.Y/site-packages/*. For building the wrapper, the GNU C++ compiler and the development headers of Python and numpy need to be installed. The resulting *_TimeTagger.so* and the high-level wrapper *TimeTagger.py* relay the Time Tagger C++ interface to Python.

```
PYTHON_FLAGS="`python-config --includes --libs`"
NUMPY_FLAGS="-I`python -c \"print(__import__('numpy').get_include())\"`"
TTFLAGS="-I/usr/include/timetagger -lTimeTagger"
CFLAGS="-std=c++11 -O2 -DNDEBUG -fPIC $PYTHON_FLAGS $NUMPY_FLAGS $TTFLAGS"

g++ $CFLAGS -shared _TimeTagger.cxx -o _TimeTagger.so
```

CHAPTER

TEN

FREQUENTLY ASKED QUESTIONS

10.1 How to detect falling edges of a pulse?

On the software level, the rising and falling edges are independent channels. In the web application, these are marked explicitly. In the software libraries, the number of a falling edge channel is a negative number of the physical channel, e.g., the falling edges of the physical channel *2* correspond to the software channel *-2*. You can also use convenience method `TimeTagger.getInvertedChannel()` to find inverted channel number for your specific hardware revision.

Note: Time Taggers delivered before mid-2018 had different channel labeling scheme. For more details, please see section *Channel Number Schema 0 and 1*.

10.2 What value should I pass to an optional channel?

You can specify a special integer value explicitly, but this is not recommended. Use the predefined constant `CHANNEL_UNUSED` instead. For C++, the constant is defined in *TimeTagger.h* and is called *CHANNEL_UNUSED*. In python, it is *TimeTagger.CHANNEL_UNUSED*.

10.3 Is it possible to use the same channel in multiple measurement classes?

Yes, absolutely. All measurement objects that you create are able to access the same time tag stream and get the same event information. This is by design of our API. Every measurement runs in its own separate thread and only the power of your CPU (clock, number of cores) and memory will limit how many of them you can create. For example, in our demonstration setup that we show on trade fairs, we run about 10 simultaneous measurements on a Microsoft Surface tablet PC without a problem. Please note that the processing power required also depends on the event rate on physical channels.

10.4 How do I choose a binwidth for a histogram?

With our Time Tagger you can choose any binwidth in the range from 1 ps to more than a day, all this range is defined in 1 picosecond steps. Together with the number of bins this will define maximum time difference you will be able to measure. Such a great flexibility lets you choose a proper binwidth purely based on the requirements of your experiment.

The following list of questions may help you to identify and decide on what binwidth value to choose.

1. What is the maximal time difference you want to measure?

   ```
   histogram_span = binwidth * n_bins
   ```

   Large values of *n_bins* require more memory and you may want to trade off binwidth for the smaller n_bins in case you want to measure very long time differences. *n_bins < 1e7* are usually fine if you create measurements in MATLAB/Python/LabView/C++/C# etc. With the Time Tagger Web App, the values of *n_bins > 10000* may result in CPU load, due to transmitting larger amount of data to the browser and refreshing the plot.

2. What time resolution do you expect from your measurement?

   Smaller binwidth will give you finer time resolution of a histogram, however, keep in mind that the real resolution is defined by the uncertainty of time measurement (timing jitter), which for Time Tagger 20 is about 34 ps RMS and 10 ps RMS for Time Tagger Ultra. Also, the timing jitter of your detectors will introduce additional timing uncertainty to your measurement. Therefore, you may want to choose a binwidth that is somewhat smaller than the measurement uncertainty of your experiment. For example, with Time Tagger 20 the binwidth of >=10 ps is a good choice.

3. What signal-to-noise ratio (SNR) you would like to achieve and in what time?

   Smaller binwidth will require a longer time to accumulate the sufficient number of counts to achieve desired noise level compared to larger binwidth. This is referring to a shot-noise that is proportional to $1/sqrt(N)$ where N is a number of counts in a single bin. This is the very same concept as SNR improvement by averaging. Larger binwidths will naturally get larger counts per bin in a shorter time for the same signal rates.

CHAPTER

ELEVEN

REVISION HISTORY

11.1 V2.7.0 - 01.10.2020

Highlights

- New measurements are automatically synchronized to the hardware. All data analyzed is guaranteed to be temporal later than the measurement's initialization, start, or clear. Data coming from the internal buffer, which was acquired before the measurement was initialized, started, or cleared, will not be analyzed. Before this release, the .sync() method was required for these tasks.

Fixes and improvements

- Added a Matlab example for SynchronizedMeasurements.
- Fixed a bug in Matlab, creating synced measurements via SynchronizedMeasruements and .getTagger().
- The last datapoint from a scope measurement was is not marked as invalid any more.

11.2 V2.6.10 - 07.09.2020

Fixes and improvements

- Fixes input delay, deadtime and test signal generator for the TimeTaggerVirtual.
- Fixes getInvertedChannel with the Swabian Synchronizer and with Time Tagger Ultra 8 devices with the old channel numbering schema.
- x axis is zoomable with Scope measurement.
- Better error handling for non-existent files with TimeTaggerVirtual and FileReader.

Python

- Changed the constants CoincidenceTimestamp_ to a Python enum (e.g., CoincidenceTimestamp_First is now CoincidenceTimestamp.First).

Matlab

- Enum for timestamp argument for Coincidence(s) is avaiable via TTCoincidenceTimestamp.

Linux

- Fix for slow Linux device opening.

11.3 V2.6.8 - 21.08.2020

Highlights

- Support for the Time Tagger Value edition. This is an upgradeable and cost-efficient version of the Time Tagger Ultra for applications with moderate timing precision requirements.

Webapp

- Added *Histogram2D* to the measurement list.
- Improved performance and responsiveness for large datasets.
- 32-bit version of the Web Application works again.
- Fixed a bug that data of stopped measurements could not be saved.
- Fixed a bug that settings saved had the file extension .json instead of .ttconf ending.
- Fixed a bug when using falling edges for Time Tagger starting with channel 0.

Python

- Fixed a bug that some named arguments could not be used anymore.

API

- Added the method *SynchronizedMeasurements.unregisterMeasurement()* to remove measurements from *SynchronizedMeasurements*.

Backend

- Improved performance of the FileWriter, exceeding 100 M tags/s on high-end CPUs.
- Improved binning performance of all histogram measurements: Correlation, FLIM, Histogram, StartStop, TimeDifferences, TimeDifferencesND.
- Fixes a deadlock in the virtual Time Tagger if a measurement accesses some public methods of the Time Tagger.

11.4 V2.6.6 - 10.07.2020

Highlights

- Swabian Synchronizer support. The Synchronizer hardware can combine 8 Time Tagger Ultras with up to 144 channels. The combined Time Tagger can be interfaced the very same as it would be only one device.
- Support for custom measurements in Python. Please see the provided programming example in the installation folder for further details.

Webapp

- Support for the Synchronizer
- Showing error messages from setLogger API in a modal window
- Load/save settings is now supported for the Time Tagger Virtual

Time Tagger Ultra

- Hardware revision 1.1 now with the same performance enhancement of 500 MHz maximum sync rate, 2ns dead time and better phase stability, as introduced before for Hardware revision > 1.1
- Dropped support for the very first Time Tagger Ultras, an error will be shown on initialization - free exchange program available

- More intuitive byte order of the bitmask in setLED
- Small modifications to the hardware channel to channel delay

Backend

- Coincidence and Coincideces have an optional parameter to select which timestamp should be inserted, the last/first completing the coincidence, the average of the event timestamps, or the first of the coincidence list.
- Fixed .net/Matlab/LabVIEW wrappers for data with empty 2D or 3D arrays
- Provide a globally registered .NET publisher policy for C#, avoiding the 'wrong dll version' message in Labview when updating the Time Tagger software
- setConditionalFilter throws an exception when invalid arguments are applied
- Hide the warning on fetching the TimeTaggerVirtual license without an internet connection
- DelayedChannel supports a negative delay
- Performance enhancements in StartStop

11.5 V2.6.4 - 27.05.2020

WebApp

- Option to enable logarithmic y-axis scaling for Counter, Histogram, HistogramLogBins and Correlation
- Redesign "Create measurement" dialog with links to the online documentation
- Fixed flickering when switching between plots
- Fixed plotting wrong data range when changing the number of data points
- Added the basic functionality of the TimeTaggerVirtual (test signal only)

New features and improvements

- Added the test signal to TimeTaggerVirtual
- Support for Ubuntu 20.04 and CentOS 8
- LabVIEW example for FileWriter and FileReader
- Improved Matlab API for VirtualTimeTagger, adding optional parameters
- Make the data transfer size configureable by .setStreamBlockSize
- Performance improvements for HistogramLogBins
- Slightly improved timing jitter at large time differences for the Time Tagger 20
- Time Tagger Application works again with 32 bit operating systems
- Connection errors are shown in the Matlab console or can be handled with the new logger functionality
- Added custom logger examples for Matlab/Python/C#

Changes

- Updated the USB library
- Stop measurements when freeTimeTagger is called (e.g. closes files on dump, isRunning now returns false)
- Reduced polling rate (0.1s) for USB reconnections

API changes

- Added .setLogger() to attach a callback function for custom info/error logging
- Rename of enumeration ErrorLevel to LogLevel
- Rename of log level constants and with new corresponding integer values

11.6 V2.6.2 - 10.03.2020

Highlights

- TimeTaggerVirtual, FileWriter, and FileReader have reached a stable state
- Improved Linux support (documentation, compiling custom Python wrappers)

New features

- Added setInputDelay, setDeadtime, getOverflows, and more to the TimeTaggerVirtual
- Add an optional parameter in setConditionalFilter for disabling the hardware delay compensation
- Infinite dumping in Dump for negative max_count
- Create a freeAllTimeTagger() method, which is called by Python atexit
- Reimplement SynchronizedMeasurements as a proxy tagger object, which auto registers new measurements without starting them
- The new SynchronizedMeasurements.isRunning() method returns if any measurement is still running
- Python: Distribute the generated C++ wrapper source for supporting future Python revisions
- C++: New IteratorBase.getLock method returning a std::unique_lock
- C++: Improved exception handling for custom measurements: exceptions now stop the measurement, runSynchronized forwards exceptions to the caller

API changes

- TimeTagger.getVersion return value is changed to a string
- C++: Use 64 bit integers for the dimensions in the array_out helpers
- C++: Rename the base class for custom measurements from _Iterator to IteratorBase
- C++: Constructors of custom measurements shall call finishInitialization instead of IteratorBase.start
- Python 2.7: Update the numpy C headers to 1.16.1

Examples and documentation

- Improved Histogram2D example
- Clearify setInputDelay vs DelayedChannel

Bug fixes

- Relax the voltage supply check in the Time Tagger Ultra hardware revision 1.4
- Use a 1 MB buffer for Dump, FileWriter, and FileReader to achieve full speed especially on network devices
- Fix getTimeTaggerModel on an active device
- Fix deadlock within sync() while the device is disconnected
- Provide the documentation on Linux
- Several fixes and improvements for the FileWriter and TimeTaggerVirtual

WebApp

- Improved default names for measurements
- Not relying on data stored within the browser any more
- Disabling mouse scrolling within numeric inputs
- Various buxfixes

11.7 V2.6.0 - 23.12.2019

Highlights

- FileWriter: New space-efficient file writer for storing time tag stream on a disk. The file size is reduced by a factor of 4 to 8. Replaces the Dump function.
- Virtual Time Tagger allows to replay previously dumped events back into the Time Tagger software engine.
- Improved behavior in the overflow mode. The hardware now also reports the amount of missed events per input channel and provides the start and the end timestamps of the overflow interval.
- New tutorial on how to implement the data acquisition for a confocal microscope
- New measurement Histogram2D for 2-dimensional histogramming with examples
- Web App: Selectable input units (s/ms/µs/ps) instead of ps only

Known issues

- FileWriter and FileReader have a low performance on network devices

API changes

- deprecated TimeTagStreamBuffer.getOverflows() – use .getEventTypes() instead
- renamed HistogramLogBin.getDataNormalized() to .getDataNormalizedCountsPerPs()
- removed deprecated TimeTagger.getChannels() - use .getChannelList() instead
- removed deprecated CHANNEL_INVALID - use CHANNEL_UNUSED instead
- removed deprecated TimeTagger.setFilter() and TimeTagger.getFilter() - use .setConditionalFilter(), .getConditionalFilter(), and .clearConditionalFilter() instead
- C++: All custom measurement class constructors must be modified, such that the parameter containing the Time Tagger is of the type TimeTaggerBase. This allows for using the custom measurement within a real Time Tagger object and the Time Tagger Virtual.
- C++: The struct Tag includes the type of event and the amount of missed events. They have replaced the overflow field.
- C++/Windows: We additionally distribute binaries for the debug runtime (/MDd)
- Matlab: TimeTagger.free() is now deprecated, use .freeTimeTagger()

New features

- Web App: Normalization (counts/s) for the Counter measurement
- getConfiguration returns the current hardware configuration as a JSON string
- added g2 normalization for HistogramLogBins with getDataNormalizedG2
- improved overflow behavior for Countrate due to the missed event counters

- improved overflow handling for the g2 normalization of Correlation and HistogramLogBin
- support for Python version 3.8
- smaller latency on low data rates due to adaptive chunk sizes of <= 20 ms
- support for the Time Tagger Ultra hardware revision 1.4

Examples

- Matlab: Faster loading of events from disk for now deprecated Dump file format
- C++: Loading events from disk stored in the new data format
- Labview: Scope example, .NET version redirection
- Mathematica: Improved example
- Python: Added "Stop" button to the countrate figure.

Bug fixes

- fixed static input delay error with conditional filter enabled since v2.2.4
- added missing TimeTagger.getTestSignalDivider() method
- Scope: Fix the output if one channel has had no events
- resolve overflows after the initialization of the Time Tagger 20
- fixes an issue with wrongly sorted events on the reconfiguration of input delays
- always emit an error event on plugging an external clock source
- fixes an unlikely case when the synchronization of the external clock got lost
- the new USB driver version fixes some random data abruption
- TTU1.3: Fix a bug which may select a wrong clock source in the first 21 seconds and wrongly activated ext clock LED
- Matlab: SynchronizedMeasurements work now in Matlab, too
- different improvements within the python and C# wrappers
- LED turns off and not red after freeing a Time Tagger
- Dump now releases the file handle after the end of the startFor duration
- Web App: Removed caching issues when up or downgrading the software

11.8 V2.4.4 - 29.07.2019

- reduced crosstalk between nonadjacent channels of the Time Tagger Ultra
- fixed a bug leading to high crosstalk with V2.4.2 for specific channels
- fixed a rare clock selection issue on the Time Tagger 20
- improved and more detailed documentation
- new method `Countrate.getCountsTotal()`, which returns the absolute number of events counted
- new Mathematica quickstart example
- new `Scope` example for LabVIEW
- support of the Time Tagger 20 series with hardware revision 2.3

- release the Python GIL while in the Time Tagger engine code
- fixed a bug in `ConstantFractionDiscriminator`, which could cause that no virtual tags were generated

11.9 V2.4.2 - 12.05.2019

- support of the Time Tagger Ultra series with hardware revision 1.3
- improve performance of short pulse sequences on the Time Tagger 20 series
- improve overflow behavior at too high input data rates
- fix the name of the 'SynchronizedMeasurements' measurement class

11.10 V2.4.0 - 10.04.2019

Libraries

- 32 bit C++ library added
- C++ and .NET libraries renamed and registered globally

API

- virtual constant fraction discriminator channel 'ConstantFractionDiscriminator' added
- 'TimeDifferenceND' added for multidimensional time differences measurements
- faster binning in 'TimeDifferences' and 'Correlation' measurements
- improved memory handling for 'TimeTageStream'
- improved Python library include
- fixed '.getNormalizedData' for 'Correlation' measurements
- various minor bug fixes and improvements

Examples

- LabVIEW project for 32 and 64 bit
- improved LabVIEW examples

Time Tagger Ultra

- 10 MHz EXT input clock detection enabled
- internal buffer size can be increased from 40 MTags to 512 MTags with 'setHardwareBufferSize'
- reduced crosstalk and timing jitter
- increased maximum transfer rate to above 65 MTags/s (Intel 5 GHz CPU on 64 bit)
- various performance improvements
- reduced deadtime to 2 ns on hardware revision >= 1.2

Time Tagger 20

- 166.6 MHz EXT input clock detection enabled

Operating systems

- equivalent support for Windows 32 and 64 bit, Ubuntu 16.04 and 18.04 64 bit, CentOS 7 64 bit

11.11 V2.2.4 - 29.01.2019

- fix the conditional filter with filter and trigger events arriving within one clock cycle
- fix issue with negativ input delays
- calling .stop() while dumping data stops the dump and closes the file
- fix device selection on reconnection after transfer errors
- synchronize tags of falling edges to their raising ones

11.12 V2.2.2 - 13.11.2018

- Removed not required Microsoft prerequisites.
- 32 bit version available

11.13 V2.2.0 - 07.11.2018

General improvements

- support for devices starting with channel 1 instead of 0
- under certain circumstances, the crosstalk for the Time Tagger 20 of channel 0-2, 0-3, 1-2, and 1-3 was highly increased, which has been fixed now
- updated and extended examples for all programming languages (Python, Matlab, C#, C++, LabVIEW)
- C++ examples for Visual Studio 2017, with debug support
- documentation for virtual channels
- Web app included in the 32 bit installer
- Linux package available for Ubuntu 16.04
- Support for Python 3.7

API

- 'HistogramLogBin' allows analyzing incoming tags with logarithmic bin sizes.
- 'FrequencyMultiplier' virtual channel class for upscaling a signal attached to the Time Tagger. This method can be used as an alternative to the 'Conditonal Filter'.
- 'SynchronizedMeasurements' class available to fully synchronize start(), stop(), clear() of different measurements.
- Second parameter from 'setConditionalFilter' changed from 'filter' to 'filtered'.

Web application

- full 'setConditionalFilter' functionality available from the backend within the Web application

11.14 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

11.15 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

11.16 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

11.17 V2.1.0 - 06.03.2018

Time Tagger Ultra

- efficient buffering of up to 60 MTags within the device to avoid overflows

11.18 V2.0.4 - 01.02.2018

Bug fixes

- Closing the web application server window works properly now

11.19 V2.0.2 - 17.01.2018

Improvements

- Matlab GUI example added
- Matlab dump/load example added

Bug fixes

- dump class writing tags multiple times when the optional channel parameter is used
- Counter and Countrate skip the time in between a .stop() and a .start() call
- The Counter class now handles overflows properly. As soon as an overflow occurs the lost data junk is skipped and the Counter resumes with the new tags arriving with no gap on the time axis.

11.20 V2.0.0 - 14.12.2017

Release of the Time Tagger Ultra

Note: The input delays might be shifted (up to a few hundred ps) compared to older driver versions.

Documentation changes

- new section 'In Depth Guides' explaining the hardware event filter

Webapp

- fixed a bug setting the input values to 0 when typing in a new value
- new server launcher screen which stops the server reliably when the application is closed

11.21 V1.0.20 - 24.10.2017

Virtual Channels

- DelayedChannel clones and optionally delays a stream of time tags from an input channel
- GatedChannel clones an input stream, which is gated via a start and stop channel (e.g. rising and falling edge of another physical channel)

API

- startFor(duration) method implemented for all measurements to acquire data for a predefined duration
- getCaptureDuration() available for all measurements to return the current capture duration
- getDataNormalized() available for Correlation
- setEventDivider(channel, divider) also transmits every nth event (divider) on channel defined

Webapp

- label for 0 on the x-axis is now 0 instead of a tiny value

C++ API:

- internal change so that clear_impl() and next_impl() must be overwritten instead of clear() and next()

Other bug fixes/improvements

- improved documentation and examples

11.22 V1.0.6 - 16.03.2017

Web application (GUI)

- load/save settings available for the Time Tagger and the measurements
- correct x-axis scaling
- input channels can be labeled
- save data as tab separated output file (for Matlab, Excel, ... import)
- fixed: saving measurement data now works reliably

- fixed: 'Initialize' button of measurements works now with tablets and phones

API

- direct time stream access possible with new class TimeTagStream (before the stream could be only dumped with Dump)
- Python 3.6 support
- better error handling (throwing exceptions) when libraries not found or no Time Tagger attached
- setTestSignal(...) can be used with a vector of channels instead of a single channel only
- Dump(...) now with an optional vector of channels to explicitly dump the channels passed
- CHANNEL_INVALID is deprecated - use CHANNEL_UNUSED instead
- Coincidences class (multiple Coincidences) can be used now within Matlab/LabVIEW

Documentation changes

- documentation of every measurement now includes a figure
- update and include web application in the quickstart section

Other bug fixes/improvements

- no internal test tags leaking through from the initialization of the Time Tagger
- Counter class not clearing the data buffer in time when no tags arrive
- search path for bitfile and libraries in Linux now work as they should
- installer for 32 bit OS available

11.23 V1.0.4 - 24.11.2016

Hardware changes

- extended event filter to multiple conditions and filter channels
- improved jitter for channel 0
- channel delays might be different from the previous version (< 1 ns)

API changes

- new function setConditionalFilter allows for multiple filter and event channels (replaces setFilter)
- Scope class implements functionality to use the Time Tagger as a 50 GHz digitizer
- Coincidences class now can handle multiple coincidence groups which is much faster than multiple instances of Coincidence
- added examples for C++ and .net

Software changes

- improved GUI (Web application)

Bug fixes

- Matlab/LabVIEW is not required to have the Visual Studio Redistributable package installed

11.24 V1.0.2 - 28.07.2016

Major changes:

- LabVIEW support including various example VIs
- Matlab support including various example scripts
- .net assembly / class library provided (32 and 64 bit)
- WebApp graphical user interface to get started without writing a single line of code
- Improved performance (multicore CPUs are supported)

API changes:

- reset() function added to reset a Time Tagger device to the startup state
- getOverflowsAndClear() and clearOverflows() introduced to be able to reset the overflow counter
- support for python 3.5 (32 and 64 bit) instead of 3.4

11.25 V1.0.0

initial release supporting python

11.26 Channel Number Schema 0 and 1

The Time Taggers delivered before mid 2018 started with channel number 0, which is very convenient for most of the programming languages.

Nevertheless, with the introduction of the Time Tagger Ultra and negative trigger levels, the falling edges became more and more important, and with the old channel schema, it was not intuitive to get the channel number of the falling edge.

This is why we decided to make a profound change, and we switched to the channel schema which starts with channel 1 instead of 0. The falling edges can be accessed via the corresponding negative channel number, which is very intuitive to use.

	Time Tagger 20 and Ultra 8		Time Tagger Ultra 18		Schema
	rising	falling	rising	falling	
old	0 to 7	8 to 15	0 to 17	18 to 35	`TT_CHANNEL_NUMBER_SCHEME_ZERO`
new	1 to 8	-1 to -8	1 to 18	-1 to -18	`TT_CHANNEL_NUMBER_SCHEME_ONE`

With release V2.2.0, the channel number is detected automatically for the device in use. It will be according to the labels on the device.

In case another channel schema is required, please use `setTimeTaggerChannelNumberScheme(int scheme)` before the first Time Tagger is initialized. If several devices are used within one instance, the first Time Tagger initialized defines the channel schema.

`int getInvertedChannel(int channel)` was introduced to get the opposite edge of a given `channel` independent of the channel schema.

INDEX

H

I

R

S

T

U

W