



# **Time Tagger User Manual**

***Release 1.2.3-local-build***

**Swabian Instruments**

**May 04, 2021**



# CONTENTS

<b>1</b>	<b>Getting Started</b>	<b>1</b>
1.1	Web Application . . . . .	1
1.2	Python . . . . .	2
1.3	LabVIEW (via .NET) . . . . .	4
1.4	Matlab (wrapper for .NET) . . . . .	4
1.5	Wolfram Mathematica (via .NET) . . . . .	4
1.6	.NET . . . . .	4
1.7	C# . . . . .	4
1.8	C++ . . . . .	5
<b>2</b>	<b>Installation instructions</b>	<b>7</b>
2.1	Requirements . . . . .	7
2.1.1	Operating System . . . . .	7
2.1.2	Installation . . . . .	7
2.1.3	Web Application . . . . .	7
2.1.4	Programming Examples . . . . .	7
<b>3</b>	<b>Tutorials</b>	<b>9</b>
3.1	Confocal Fluorescence Microscope . . . . .	9
3.1.1	Time Tagger configuration . . . . .	10
3.1.2	Intensity scanning microscope . . . . .	11
3.1.3	Fluorescence Lifetime Microscope . . . . .	12
3.1.4	Alternative pixel trigger formats . . . . .	13
<b>4</b>	<b>Synchronizer</b>	<b>17</b>
4.1	Overview . . . . .	17
4.2	Requirements . . . . .	17
4.3	Cable connections . . . . .	17
4.3.1	Using an external reference clock . . . . .	19
4.4	Software and channel numbering . . . . .	19
4.4.1	Incomplete cable connections . . . . .	20
4.4.2	Buffer overflows . . . . .	20
4.5	Limitations . . . . .	20
4.5.1	Conditional filter . . . . .	20
4.5.2	Internal test signal . . . . .	20
4.6	Status LEDs and troubleshooting . . . . .	21
<b>5</b>	<b>Hardware</b>	<b>23</b>
5.1	Input channels . . . . .	23
5.1.1	Electrical characteristics . . . . .	23

5.1.2	High Resolution Mode (beta)	23
5.2	Data connection	24
5.3	Status LEDs	24
5.4	Test signal	24
5.5	Virtual channels	25
5.6	Synthetic input delay	25
5.7	Synthetic dead time	25
5.8	Conditional Filter	25
5.9	Bin equilibration	26
5.10	Overflows	26
5.11	External Clock Input	26
5.12	Synchronization signals - Time Tagger Ultra only	27
5.13	General purpose IO (GPIO) - Time Tagger Ultra only	27
5.14	General purpose IO (GPIO) - Time Tagger 20 only	27
<b>6</b>	<b>Software Overview</b>	<b>29</b>
6.1	Web application	29
6.2	Precompiled libraries and high-level language bindings	29
6.3	C++ API	29
<b>7</b>	<b>Application Programmer's Interface</b>	<b>31</b>
7.1	Examples	31
7.1.1	Measuring cross-correlation	31
7.1.2	Using virtual channels	32
7.1.3	Using multiple Time Taggers	32
7.2	The TimeTagger Library	33
7.2.1	Units of measurement	33
7.2.2	Channel numbers	33
7.2.3	Unused channels	33
7.2.4	Constants	33
7.2.5	Enums	33
7.2.6	Functions	35
7.3	The TimeTagger class	36
7.4	The TimeTaggerVirtual class	44
7.5	Virtual Channels	45
7.5.1	Available virtual channels	45
7.5.2	Common methods	45
7.5.3	Combiner	46
7.5.4	Coincidence	46
7.5.5	Coincidences	47
7.5.6	FrequencyMultiplier	48
7.5.7	GatedChannel	48
7.5.8	DelayedChannel	49
7.5.9	ConstantFractionDiscriminator	50
7.5.10	EventGenerator	50
7.5.11	TriggerOnCountrate	51
7.6	Measurement Classes	52
7.6.1	Available measurement classes	53
7.6.2	Common methods	54
7.6.3	Event counting	54
7.6.4	Time histograms	58
7.6.5	Fluorescence-lifetime imaging (FLIM)	67
7.6.6	Time-tag-streaming	73
7.6.7	Helper classes	79

7.6.8	Custom Measurements . . . . .	81
<b>8</b>	<b>In Depth Guides</b>	<b>83</b>
8.1	Conditional Filter . . . . .	83
8.1.1	Example configurations . . . . .	83
8.1.2	Understanding the filtering mechanism . . . . .	86
8.1.3	Setup of the Conditional Filter . . . . .	88
8.2	Raw Time-Tag-Stream access . . . . .	89
8.2.1	Dumping and post-processing . . . . .	90
8.2.2	On-the-fly processing . . . . .	90
8.3	Synchronization of the Time Tagger pipeline . . . . .	91
<b>9</b>	<b>Linux</b>	<b>93</b>
<b>10</b>	<b>Frequently Asked Questions</b>	<b>95</b>
10.1	How to detect falling edges of a pulse? . . . . .	95
10.2	What value should I pass to an optional channel? . . . . .	95
10.3	Is it possible to use the same channel in multiple measurement classes? . . . . .	95
10.4	How do I choose a binwidth for a histogram? . . . . .	96
<b>11</b>	<b>Revision History</b>	<b>97</b>
11.1	V2.8.4 - 04.05.2021 . . . . .	97
11.2	V2.8.2 - 26.04.2021 . . . . .	97
11.3	V2.8.0 - 29.03.2021 . . . . .	97
11.4	V2.7.6 - 26.04.2021 . . . . .	98
11.5	V2.7.4 - 19.04.2021 . . . . .	98
11.6	V2.7.2 - 22.12.2020 . . . . .	98
11.7	V2.7.0 - 01.10.2020 . . . . .	99
11.8	V2.6.10 - 07.09.2020 . . . . .	100
11.9	V2.6.8 - 21.08.2020 . . . . .	100
11.10	V2.6.6 - 10.07.2020 . . . . .	101
11.11	V2.6.4 - 27.05.2020 . . . . .	101
11.12	V2.6.2 - 10.03.2020 . . . . .	102
11.13	V2.6.0 - 23.12.2019 . . . . .	103
11.14	V2.4.4 - 29.07.2019 . . . . .	105
11.15	V2.4.2 - 12.05.2019 . . . . .	105
11.16	V2.4.0 - 10.04.2019 . . . . .	105
11.17	V2.2.4 - 29.01.2019 . . . . .	106
11.18	V2.2.2 - 13.11.2018 . . . . .	106
11.19	V2.2.0 - 07.11.2018 . . . . .	106
11.20	V2.1.6 - 17.05.2018 . . . . .	107
11.21	V2.1.4 - 21.03.2018 . . . . .	107
11.22	V2.1.2 - 14.03.2018 . . . . .	107
11.23	V2.1.0 - 06.03.2018 . . . . .	107
11.24	V2.0.4 - 01.02.2018 . . . . .	107
11.25	V2.0.2 - 17.01.2018 . . . . .	108
11.26	V2.0.0 - 14.12.2017 . . . . .	108
11.27	V1.0.20 - 24.10.2017 . . . . .	108
11.28	V1.0.6 - 16.03.2017 . . . . .	109
11.29	V1.0.4 - 24.11.2016 . . . . .	109
11.30	V1.0.2 - 28.07.2016 . . . . .	110
11.31	V1.0.0 . . . . .	110
11.32	Channel Number Schema 0 and 1 . . . . .	110
	<b>Index</b>	<b>113</b>



## 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.

Note: The Web Application has the port 50120 as default port. If this collides with another application you can change the port with passing the argument `TimeTaggerServer.exe -p 50120`.

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 Number of data points 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 be displayed. You can zoom in using the controls on the plot.

4. 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 (especially the server window) 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 **hello\_world.py** script by entering `run hello_world`

The *hello\_world* executes a simple yet useful measurement that demonstrates many essential features of the *Time Tagger* programming interface: #. Connect your Time Tagger #. Start the built-in test signal (~0.8 MHz square wave) and apply it to channels 1 and 2 #. Control the trigger level of your inputs - although it is not necessary here #. Initialize a standard measurement (*Correlation*) in order to find the delay of the test signal between channel 1 and 2 #. How to control the delay of different inputs programmatically.

You are encouraged to open and read the `hello_world.py` file in an editor to understand what it is doing. With this basic knowledge, you can explore the other examples in the 1-Quickstart folder:



No.	Topic	Classes & Methods
<b>Basic software control</b> (folder 1-basic_software_control)		
1-A	Create a measurement	<code>createTimeTagger()</code> , <code>Counter.getData()</code> , <code>Counter.getIndex()</code>
	Count rate trace	<code>Counter</code>
1-B	Start & stop measurements	<code>Countrate</code> , <code>start()</code> , <code>stop()</code> , <code>startFor()</code>
1-C	Synchronize measurements	<code>SynchronizedMeasurements</code>
	Use different histograms	<code>Correlation</code> , <code>Histogram</code> , <code>StartStop</code> , <code>HistogramLogBins</code>
1-D	Virtual Channels	<code>DelayedChannel</code> , <code>Coincidence</code> , <code>GatedChannel</code>
1-E	Logging errors	<code>setLogger()</code>
<b>Controlling the hardware</b> (folder 2-controlling-the-hardware)		
2-A	Get hardware information	<code>scanTimeTagger()</code> , <code>TimeTagger.getSerial()</code> ,
		<code>TimeTagger.getModel()</code> , <code>TimeTagger.getSensorData()</code> ,
		<code>TimeTagger.getConfiguration()</code>
2-B	The input trigger level	<code>TimeTagger.setTriggerLevel()</code> , <code>TimeTagger.getDACRange()</code>
2-C	Filter tags on hardware	<code>TimeTagger.setConditionalFilter()</code> , <code>TimeTagger.setEventDivider()</code>
2-D	Control input delays	<code>TimeTagger.setInputDelay()</code> , <code>Histogram2D</code>
2-E	Overflows	<code>TimeTagger.getOverflows()</code> , <code>TimeTagger.setTestSignalDivider()</code>
2-F	HighRes mode	<code>createTimeTagger()</code> , <code>TimeDifferences</code>
<b>Dump and re-analyze time-tags</b> (folder 3-dump-and-reanalyze-time-tags)		
3-A	Dump tags by FileWriter	<code>FileWriter</code>
3-B	The Time Tagger Virtual	<code>createTimeTaggerVirtual()</code> , <code>TimeTaggerVirtual</code>
<b>Working with raw time-tags</b> (folder 4-working-with-raw-time-tags)		
4-A	The FileReader	<code>FileReader</code> , <code>TimeTagStreamBuffer</code>
4-B	Streaming raw time-tags	<code>TimeTagStream</code>
4-C	Custom Measurements	<code>CustomMeasurement</code>

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, 64 bit and CIL) 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 Visual Studio C# project provided in the `.\examples\csharp\Quickstart` directory covers the basics of how to use the Time Tagger .NET API. An example of creating 'custom measurements' is also included.

Please copy the project folder to a directory within the user environment such that files can be written within the directory.

An 'Example Suite' is provided in the `.\examples\csharp\ExampleSuite` directory. 'Example Suite' is an interactive application that demonstrates various measurements that can be performed with the `TimeTagger`. Reference source code to setup and plot (with `OxyPlot`) each measurement is also provided within the application. Additionally,

the application contains examples for creating and using ‘*Virtual channels*’, ‘*Filtering*’ and ‘*Accessing the raw time tags*’.

---

**Note:**

- To run the Example Suite ‘.NET Core 3.1 Desktop Runtime (v3.1.10)’ is required.
- 

## 1.8 C++

The provided Visual Studio C++ project can be found in `.\examples\cpp\`. Using the C++ interface is the most performant way to interact with the Time Tagger as it supports writing custom measurement classes with no overhead. 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 v142 or newer Visual Studio runtime (/MD)
-



## 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.



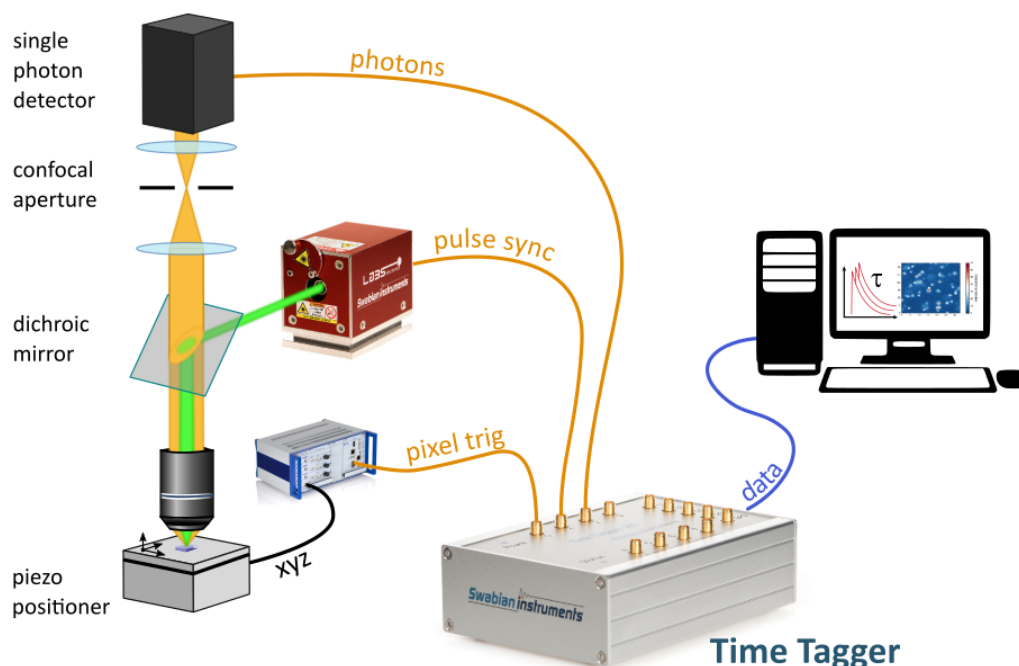
## TUTORIALS

### 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.

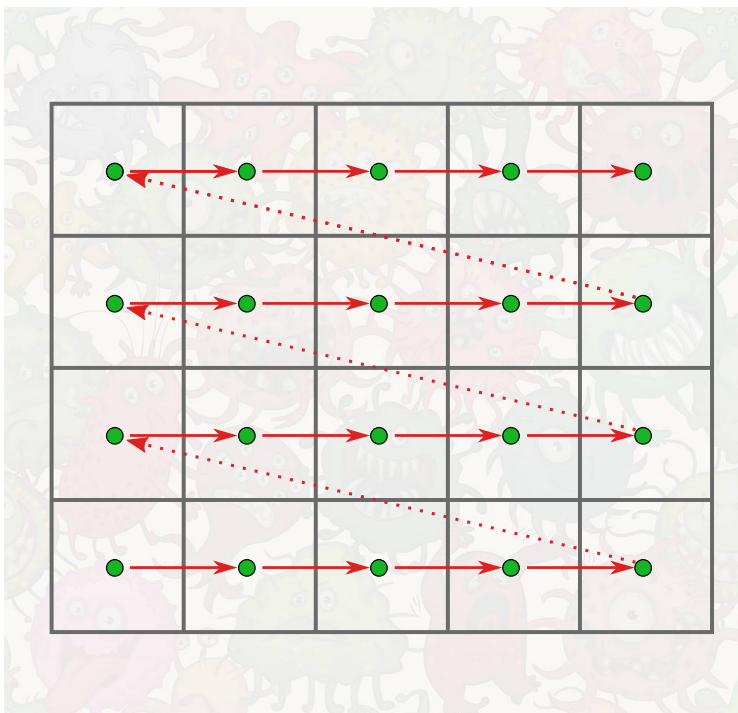
The following drawing shows a typical confocal fluorescence microscope setup.



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 *TimeDifferences* measurement from the Time Tagger library. In this subsection, we will use the *TimeDifferences* measurement.

The *TimeDifferences* 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 = TimeDifferences(
    tt,
    click_channel=SPD_CH,
    start_channel=LASER_CH,
    next_channel=PIXEL_START_CH,
    binwidth=binwidth,
    n_bins=n_bins,
    n_histograms=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)
```

(continues on next page)

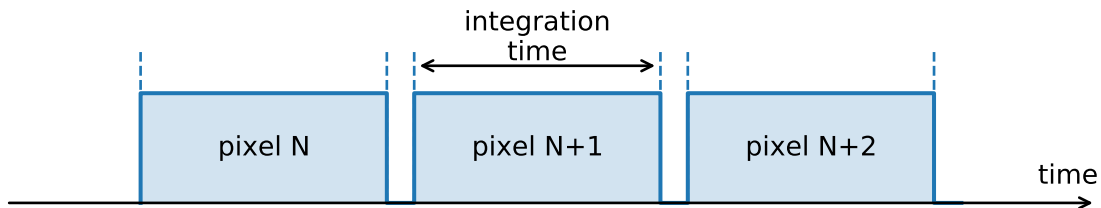
(continued from previous page)

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

### 3.1.4 Alternative pixel trigger formats

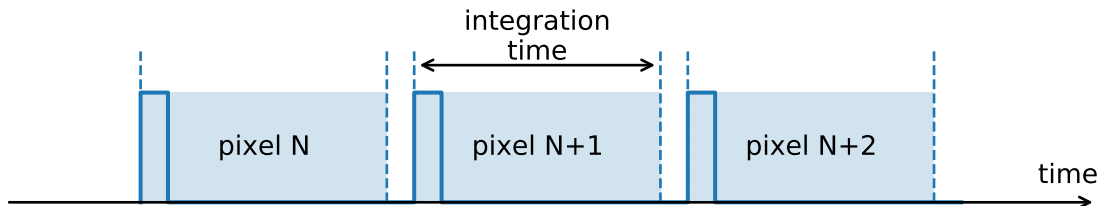
What if a piezo-scanner provides a different trigger signal compared to considered in the previous sections? In this section, we look into a few common types of trigger signals and how to adapt our data acquisition to make them work.

#### 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.

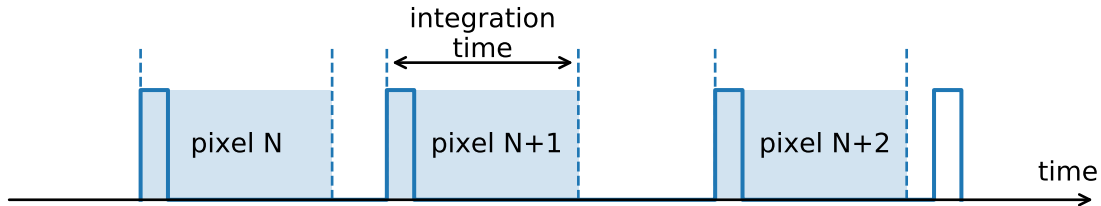
```
integr_time = int(1e10) # Integration time of 10 ms in picoseconds
delayed_vch = DelayedChannel(tt, PIXEL_START_CH, integr_time)
PIXEL_END_CH = delayed_vch.getChannel()

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 *TimeDifferences* measurement 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.

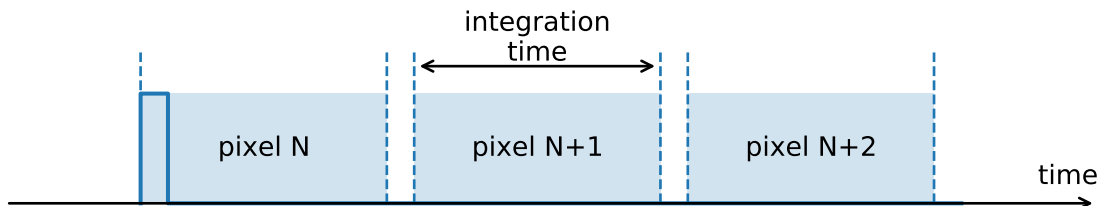
The Time Tagger library provides you with the necessary tools to enforce a fixed integration time when using the *TimeDifferences* measurement. Gating the detector events can be done with the *GatedChannel*. The example code is provided below.

```
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 = TimeDifferences(tt,
    click_channel=GATED_SPD_CH,
    start_channel=LASER_CH,
    next_channel=PIXEL_START_CH,
    binwidth=binwidth,
    n_bins=n_bins,
    n_histograms=n_pixels
)
```

### Line pulse but no pixel pulses



When a scanning system only has the line-start signal and does not provide the pixel pulses, we have to define time intervals for each pixel by other means. The pixel markers can be easily generated with *EventGenerator* virtual channel which generates events at times relative to the trigger event. Furthermore, the *EventGenerator* allows you to generate not only pixel markers that are equally spaced but also pixels that are spaced non-uniformly or have time varying integration times. For instance, you will find the *EventGenerator* particularly powerful, if you work with resonant galvo-scanners and need to correct integration time and pixel spacing according to the speed profile of your scanner. The example below shows how to apply *EventGenerator* for generation of pixel markers.

```

nx_pix = 300          # Number of pixels along x-axis
ny_pix = 200          # Number of pixels/lines along y-axis
integr_time = int(3e9) # Integration time of 3 ms in picoseconds
line_duration = 1e12  # Duration of the line scan in picoseconds
binwidth = 50         # in picoseconds
n_bins = 2000         # number of bins in a histogram
n_pixels = nx_pix * ny_pix # number of histograms

LINE_START_CH = 3

# Pixels are equally spaced in time (constant speed)
pixel_start_times = numpy.linspace(0, line_duration, nx_pix)
# Pixel integration time is constant
pixel_stop_times = pixel_start_times + integr_time

# Create EventGenerator channels
pixel_start_vch = EventGenerator(tt, LINE_START_CH, pixel_start_times)
pixel_stop_vch = EventGenerator(tt, LINE_START_CH, pixel_stop_times)

PIXEL_START_CH = pixel_start_vch.getChannel()
PIXEL_END_CH = pixel_stop_vch.getChannel()

# Use GatedChannel to gate the detector
gated_vch = GatedChannel(tt, SPD_CH, PIXEL_START_CH, PIXEL_END_CH)
GATED_SPD_CH = gated_vch.getChannel()

flim = TimeDifferences(
    tt,
    click_channel=GATED_SPD_CH,
    start_channel=LASER_CH,
    next_channel=PIXEL_START_CH,
    binwidth=binwidth,
    n_bins=n_bins,
    n_histograms=n_pixels
)

```

**Note:** In the TimeTagger software v2.7.2 we have completely redesigned *Flim* measurement. It support easy interface similar to *TimeDifferences*, as well as high-performance frame streaming interface that allows for real-time video-rate FLIM imaging.



## 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](http://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.

---

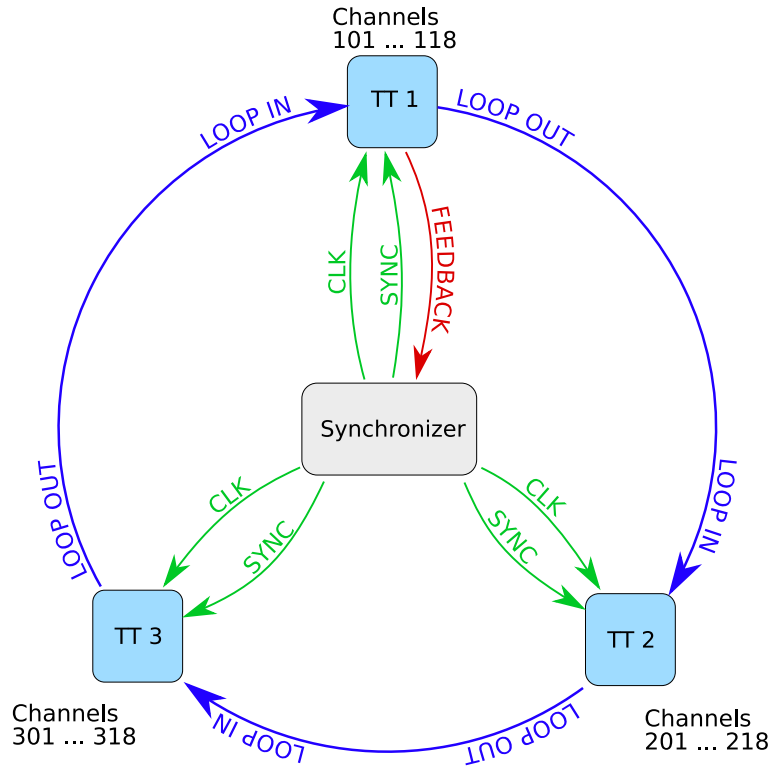
**Note:** After the release of the Synchronizer, we have changed the connector labels on the front panel of Time Tagger Ultra. In this section, we use the new labeling scheme, while showing the corresponding old labels in brackets: *NEW\_LABEL (OLD\_LABEL)*.

---

Table 1: Connections between the Synchronizer and Time Taggers

Synchronizer	Time Tagger	Description
<i>CLK OUT</i> <N>	<i>CLK IN (CLK)</i>	500 MHz clock
<i>SYNC OUT</i> <N>	<i>SYNC IN (AUX IN 1)</i>	Synchronization data
<i>FDBK IN</i>	<i>FDBK OUT (AUX OUT 2)</i>	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

```
CHANNEL_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](#).



## 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

#### 5.1.2 High Resolution Mode (beta)

The Time Tagger Ultra Performance can operate in different High Resolution (HighRes) modes. An increased resolution is achieved by directing the signal from a single input to multiple time-to-digital conversions (TDCs). Depending on the mode, 2, 4, or 8 TDCs are used per input. By averaging the results, a single timestamp with lower jitter is generated. On the other hand, this process reduces the number of usable signal inputs.

The table shows the usable inputs for the different modes. Channels available with the minimal four-channel license are shown without parenthesis. Further channels are added from the list in parenthesis in the HighRes column first and added in the Standard resolution column if the amount of HighRes channels is exhausted.

Mode	HighRes channels	Standard channels
Standard		1 - 4, (5 - 18)
HighResA	1, 3, 5, 7, (10, 12, 14, 16)	(9, 18)
HighResB	1, 5, 10, 14	(9, 18)
HighResC	5, 14	9, 18

---

**Note:** As a result of the averaging process, the quality of the calculated timestamps is affected by relative changes of internal delays of the contributing inputs. These delays are affected especially by the temperature of the device. It is strongly recommended to let the device heat up for at least 10 s before starting a measurement. Constant average

count rates (averaged over the timescale of hundreds of milliseconds) will provide the best results. If you need more information on this topic, please contact us via [support@swabianinstruments.com](mailto:support@swabianinstruments.com).

---

## 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  $\geq 2.0$  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 as a simple way of data rate reduction. The dead time can be set individually for rising and for falling edges in each channel.

## 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.

---

**Note:** In Time Tagger 20, the software-defined input delays as set by the method `TimeTagger.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 10 or 500 MHz for the Time Tagger Ultra.

The CLK input of the Time Tagger Ultra requires 100 mVpp - 3 Vpp AC coupled into 50 Ohm, 500 mVpp are recommended. As soon as this frequency is applied to the EXT CLK input, the Time Tagger Ultra is locked to it. The lock status can be read off the LED color: If the CLK LED shines green, the Time Tagger is locked. If it shines red, a wrong frequency is applied.

---

**Note:** The external clock input is only supported for the Time Tagger Ultra, not for the Time Tagger 20.

---

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.



## 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.



## APPLICATION PROGRAMMER'S INTERFACE

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* or *CustomMeasurement* class (best performance, but only available in the C++, 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 time-tags into a file using *FileWriter* and then read the resulting file to perform desired analysis of the time-tags. This also enables to keep a record of the complete chronology of the events in your experiment.

### 7.1 Examples

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

#### 7.1.1 Measuring cross-correlation

The code below shows a simple but operational example of 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)

# Run Correlation for 1 second to accumulate the data
corr.startFor(int(1e12), clear=True)
corr.waitUntilFinished()
```

(continues on next page)

(continued from previous page)

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

### 7.1.2 Using virtual channels

Time Tagger API implements on-the-fly time-tag processing through *virtual channels*. The following example shows how time-tags 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 time-tags 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()])

# Run Countrate for 1 second and print the result for all three channels
rate.startFor(int(1e12), clear=True)
rate.waitUntilFinished()
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.

### 7.1.3 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`.

## 7.2 The TimeTagger Library

The Time Tagger Library contains classes for hardware access and data processing. This section covers the units and terminology definitions as well as describes constants and functions defined at the library level.

### 7.2.1 Units of measurement

Time is measured and specified in picoseconds. Time-tags 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.

### 7.2.2 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.

### 7.2.3 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.

### 7.2.4 Constants

#### **CHANNEL\_UNUSED**

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

### 7.2.5 Enums

#### **class Resolution**

This Enum is used to determine the resolution mode in `createTimeTagger()`. Details on the available inputs are listed in the *hardware overview*.

##### **Standard**

Use one time-to-digital conversion per channel. All physical inputs can be used.

##### **HighResA**

Use two time-to-digital conversions per channel. The resolution is increased by a factor of  $\simeq \sqrt{2}$  compared to the Standard mode, but only a reduced number of certain inputs can be used. Some inputs may remain in Standard mode depending on your license.

**HighResB**

Use four time-to-digital conversions per channel. The resolution is increased by a factor of  $\simeq 2$  compared to the Standard mode, but only a reduced number of certain inputs can be used. Some inputs may remain in Standard mode depending on your license.

**HighResC**

Use four time-to-digital conversions per channel. The resolution is increased by a factor of  $\simeq \sqrt{8}$  compared to the Standard mode, but only a reduced number of certain inputs can be used. Some inputs may remain in Standard mode depending on your license.

**class ChannelEdge**

This Enum is used to select the channels that `TimeTagger.getChannelList()` returns.

**All**

Rising and falling edges of channels with HighRes and Standard resolution.

**Rising**

Rising edges of channels with HighRes and Standard resolution.

**Falling**

Falling edges of channels with HighRes and Standard resolution.

**HighResAll**

Rising and falling of channels edges with HighRes resolution.

**HighResRising**

Rising edges of channels with HighRes resolution.

**HighResFalling**

Falling edges of channels with HighRes resolution.

**StandardAll**

Rising and falling edges of channels with Standard resolution.

**StandardRising**

Rising edges of channels with Standard resolution.

**StandardFalling**

Falling edges of channels with Standard resolution.

**class CoincidenceTimestamp**

This Enum is used to select the timestamp that is attributed to a coincidence event in `Coincidence/Coincidences`.

**Last**

Use the last time-tag to define the timestamp of the coincidence.

**Average**

Calculate the average timestamp of all time-tags in the coincidence and use it as the timestamp of the coincidence.

**First**

Use the first time-tag to define the timestamp of the coincidence.

**ListedFirst**

Use the timestamp of the channel at the first position of the list when `Coincidence` or a group of `Coincidences` is instantiated.



## 7.2.6 Functions

### **createTimeTagger** (*serial*="", *resolution*=*Resolution.Standard*)

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.

If the HighRes mode is available, it can be selected from *Resolution*. Details on the available inputs are listed in the *hardware overview*.

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

#### **Parameters**

- **serial** (*str*) – Serial number string of the device or empty string
- **resolution** (*Resolution*) – Select the resolution of the Time Tagger. The default is *Resolution.Standard*.

**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 bin-widths,

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

### **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[str]

### **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. Callback function shall have the following signature *callback(level, message)*. By default the log messages are printed into the console.

Python example:

```
def logger_func(level, message):
    print(level, message)
setLogger(logger_func)
```

Matlab example:

```
function logger_func(level, message)
    fprintf('%d : %s\n', level, message)
end
TimeTagger.setLogger(@logger_func)
```

**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** int

## 7.3 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 time-tag-stream. Every *measurement* and *virtual channel* requires a reference to the *TimeTagger* object with which it will be associated.

**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** (*int*) – Physical channel number
- **voltage** (*float*) – Trigger level in Volts

**getTriggerLevel** (*channel*)

Returns trigger level for the specified physical channel number.

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

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

**Return type** float

**setDelayHardware** (*channel*, *delay*)

**Caution:** Time Tagger Ultra only.

Set an artificial delay per *channel*. The delay can be positive or negative. This delay is applied onboard the Time Tagger directly after the time-to-digital conversion, so it affects also the *Conditional Filter*.

**Parameters**

- **channel** (*int*) – Channel number
- **delay** (*int*) – Delay time in picoseconds, the maximum/minimum value allowed is  $\pm 20000000$  ( $\pm 2 \mu\text{s}$ )

**getDelayHardware** (*channel*)

**Caution:** Time Tagger Ultra only.

Returns the value of the delay applied onboard the Time Tagger in picoseconds for the specified *channel*.

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

**Returns** Delay time in picoseconds

**Return type** `int`

**setDelaySoftware** (*channel*, *delay*)

Set an artificial delay per *channel*. The delay can be positive or negative. This delay is applied on the computer, so it does not affect onboard processes such as the Conditional Filter.

**Parameters**

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

**getDelaySoftware** (*channel*)

Returns the value of the delay applied on the computer in picoseconds for the specified *channel*.

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

**Returns** Delay time in picoseconds

**Return type** `int`

**setInputDelay** (*channel*, *delay*)

This is a convenience method. It calls `TimeTagger.setDelaySoftware()` if you use a Time Tagger 20 or the delay is  $> 2 \mu\text{s}$ , otherwise `TimeTagger.setDelayHardware()` is called.

**Parameters**

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

**getInputDelay** (*channel*)

This is a convenience method. It will return the sum of `TimeTagger.getDelaySoftware()` and `TimeTagger.getDelayHardware()`.

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

**Returns** Delay time in picoseconds

**Return type** int

**getHardwareDelayCompensation** (*channel*)

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

This compensation can be understood as an implicit part of *TimeTagger.setDelayHardware()* and *TimeTagger.setDelaySoftware()*. If your device is able to set an arbitrary delay onboard, this applies to the hardware delay compensation as well.

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

**Returns** Hardware delay compensation in picoseconds

**Return type** int

**setConditionalFilter** (*trigger, filtered, hardwareDelayCompensation=True*)

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[int]*) – List of channel numbers
- **filtered** (*list[int]*) – List of channel numbers
- **hardwareDelayCompensation** (*bool*) – optional, default: True. If set to False, the physical hardware delay will not be compensated. This is only relevant for devices without *TimeTagger.setDelayHardware()*, do not set this value to False if your device is capable of onboard delay compensation. Without onboard delay compensation, setting the value to False 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. Equivalent 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[int]

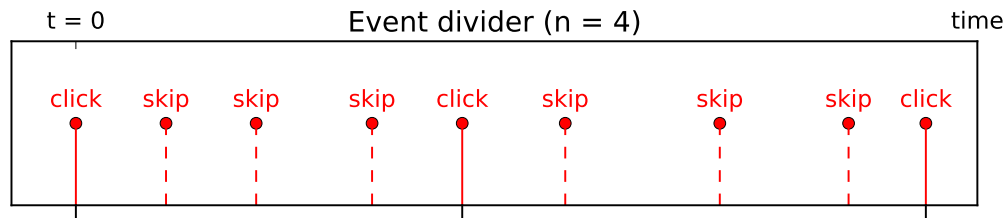
**getConditionalFilterFiltered** ()

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

**Returns** List of channel numbers

**Return type** list[int]

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

#### **getEventDivider** (*channel*)

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

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

**Returns** Divider factor value

**Return type** int

#### **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.

---

**Note:** The specified deadtime of the Time Tagger Ultra is 2.1 ns. With the default hardware deadtime filter of 2 ns, an event arriving between 2 ns and 2.1 ns after the last event might be dropped.

---

#### Parameters

- **channel** (*int*) – Channel number.
- **deadtime** (*int*) – 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** int

#### **getDeadtime** (*channel*)

Returns the dead time value for the specified *channel*.

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

**Returns** Deadtime value in picoseconds

**Return type** int

#### **setTestSignal** (*channels*, *bool state*)

Connect or disconnect the channels with the on-chip uncorrelated signal generator.

**Parameters**

- **channels** (*list[int]*) – 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** (*int*) – Physical channel number

**Returns** True/False

**Return type** bool

**getSerial** ()

Returns the hardware serial number.

**Returns** Serial number string

**Return type** str

**getModel** ()

**Returns** Model name as string

**Return type** str

**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** int

**getOverflowsAndClear** ()

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

**Returns** Number of overflows

**Return type** int

**clearOverflows** ()

Set the overflow counter to zero.

**getFence** (*alloc\_fence=True*)

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 number 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** int

**waitForFence** (*fence, timeout=-1*)

Wait for a fence in the data stream. See `getFence()` for more details.

**Parameters**

- **fence** (*int*) – fence object, which shall be waited on
- **timeout** (*int*) – 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** (*timeout*)

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. It is equivalent to `waitForFence(getFence())`.

**Parameters** **timeout** (*int*) – optional, default: -1. timeout in milliseconds. Negative means no timeout, zero returns immediately.

**Returns** True if the synchronization was successful, false on timeout

**Return type** bool

**getPcbVersion** ()

Returns Time Tagger PCB (Printed circuit board) version.

**Returns** PCB version

**Return type** str

**getDACRange** ()

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

**Returns** Min and max voltage in Volt

**Return type** (float, float)

**registerChannel** (*channel*)

Enable transmission of time tags on the specified channel.

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

**unregisterChannel** (*channel*)

Disable transmission of time tags on the specified channel.

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

**getChannelList** (*type*)

Returns a list of channels corresponding to the given *type*.

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

**Returns** List of channel numbers

**Return type** list[int]

**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** (*int*) – Channel number

**Returns** Channel number

**Return type** int

**isChannelUnused** (*channel*)

Returns true if the passed channel number is `CHANNEL_UNUSED`.

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

**Returns** True/False

**Return type** bool

**setHardwareBufferSize** (*size*)

**Caution:** Time Tagger 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** (*int*) – 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[float]

**getDistributionCount** ()

Returns the calibration data represented in counts.

**Returns** Distribution data

**Return type** 2D\_array[int]

**getDistributionPSec** ()

Returns the calibration data in picoseconds.

**Returns** Calibration data

**Return type** 2D\_array[int]

**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** int

**setStreamBlockSize** (*max\_events=131072, max\_latency=20*)

This option controls the latency and the block size of the data stream. 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 arrives for 512 ns on the Time Tagger Ultra and 1536 ns for the Time Tagger 20.

**Parameters**

- **max\_events** (*int*) – maximum number of events within one block (256 - 32M), default: 131072 events
- **max\_latency** (*int*) – maximum latency in milliseconds for constant input rates (1 to 10000), default: 20 ms.



**setTestSignalDivider** (*divider*)

Change the frequency of the on-chip test signal.

For the Time Tagger Ultra, the base frequency is 50 MHz and the default divider 63 corresponds to ~800 kCounts/s.

For the Time Tagger 20, the base frequency is 62.5 MHz and the default divider is 74 corresponds to ~850 kCounts/s.

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

**getTestSignalDivider** ()

Returns the value of test signal division factor.

**getSensorData** ()

Prints all available sensor data for the given board. The Time Tagger 20 has no onboard sensors.

**Returns** Tabulated sensor data

**Return type** str

**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** (*int*) – LED bitmask.

**setSoundFrequency** (*freq\_hz*)

Set the Time Tagger's internal buzzer to a frequency in Hz.

**Parameters** **freq\_hz** (*int*) – The sound frequency in Hz, use 0 to switch the buzzer off.

**getConfiguration** ()

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

## 7.4 The TimeTaggerVirtual class

In the TimeTagger software version 2.6.0, we have introduced the new *TimeTaggerVirtual*, which allows re-playing 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 immediately.

#### **Parameters**

- **file** (*str*) – the file to be replayed
- **begin** (*int*) – duration in ps to skip at the beginning of the file. A negative time will generate a pause in the replay.
- **duration** (*int*) – 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** int

**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** (*int*) – 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. Extreme slow replay speed between 0.0 and 0.1 is not supported.

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

**getReplaySpeed()**

Returns the current speed factor.

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

## 7.5 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 they receive time-tags from a real or another virtual channel and output to which they send processed time-tags.

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.

### 7.5.1 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.

### 7.5.2 Common methods

*VirtualChannel*.**getChannel()**

*VirtualChannel*.**getChannels()**

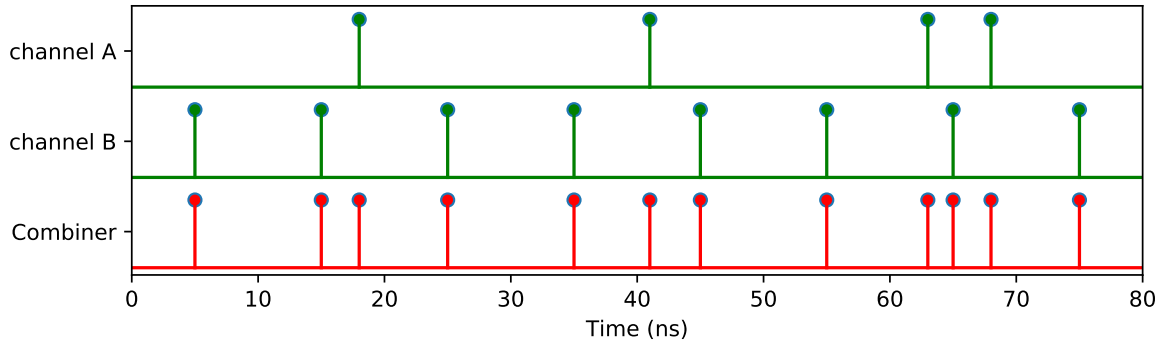
Returns the channel number(s) corresponding to the virtual channel(s). Use this channel number the very same way as the channel number of physical channel, for example, as an input to a measurement class or another virtual channel.

---

**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.

### 7.5.3 Combiner



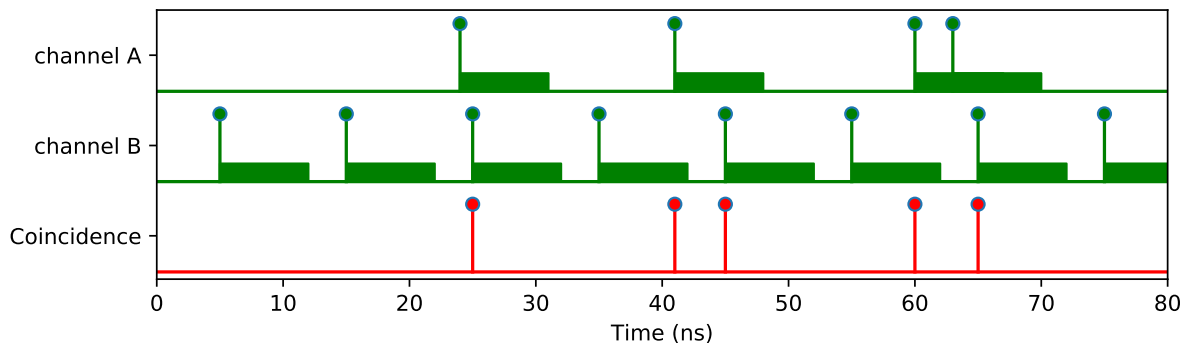
Combines two or more channels into one. The virtual channel is triggered, e.g., for two channels when either channel A OR channel B received a signal.

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

#### Parameters

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

### 7.5.4 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=1000, timestamp=CoincidenceTimestamp.Last)
```

#### Parameters

- **tagger** (*TimeTagger*) – time tagger object instance

- **channels** (*list[int]*) – list of channels on which coincidence will be detected in the virtual channel
- **coincidenceWindow** (*int*) – maximum time between all events for a coincidence [ps]
- **timestamp** (*CoincidenceTimestamp*) – type of timestamp for virtual channel

## 7.5.5 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:

```
from TimeTagger import Coincidences, Coincidences, CoincidenceTimestamp

coinc = Coincidences(tagger, [[1,2], [2,3,5]], coincidenceWindow=10000,
    ↳timestamp=CoincidenceTimestamp.ListedFirst)
coinc_chans = coinc.getChannels()
coinc1_ch = coinc_chans[0] # double coincidence in channels [1,2] with timestamp of
    ↳channel 1
coinc2_ch = coinc_chans[1] # triple coincidence in channels [2,3,5] with timestamp
    ↳of chanel 2

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

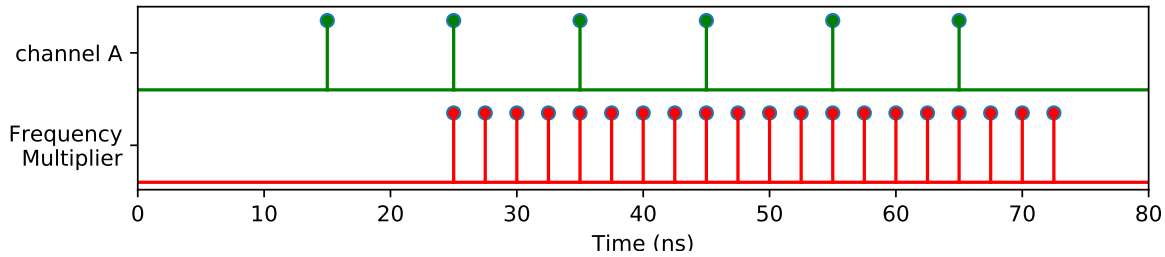
**Note:** Only C++ and python support jagged arrays (array of arrays, like `uint[][]`) which are required to combine several coincidence groups and pass them to the constructor of the *Coincidences* class. Hence, the API differs for Matlab, which requires a cell array of 1D vectors to be passed to the 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[int]]*) – list of channel groups on which coincidence will be detected in the virtual channel
- **coincidenceWindow** (*int*) – maximum time between all events for a coincidence [ps]
- **timestamp** (*CoincidenceTimestamp*) – type of timestamp for virtual channel (Last, Average, First, ListedFirst)

## 7.5.6 FrequencyMultiplier



Frequency Multiplier for a channel with a periodic signal.

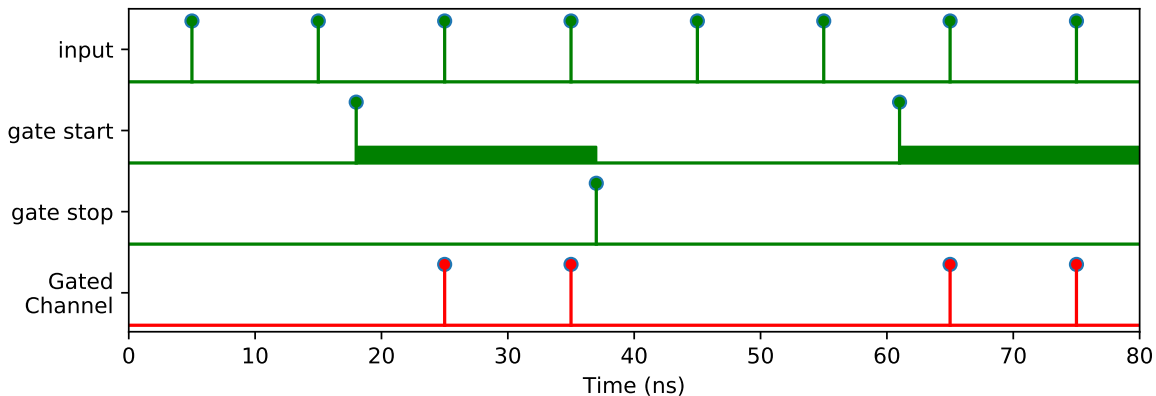
**Note:** Very high output frequencies create a high CPU load, eventually leading to *overflows*.

**class** `FrequencyMultiplier` (*tagger*, *input\_channel*, *multiplier*)

### Parameters

- **tagger** (`TimeTagger`) – time tagger object instance
- **input\_channel** (*int*) – channel on which the upscaling of the frequency is based on
- **multiplier** (*int*) – frequency upscaling factor

## 7.5.7 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`.

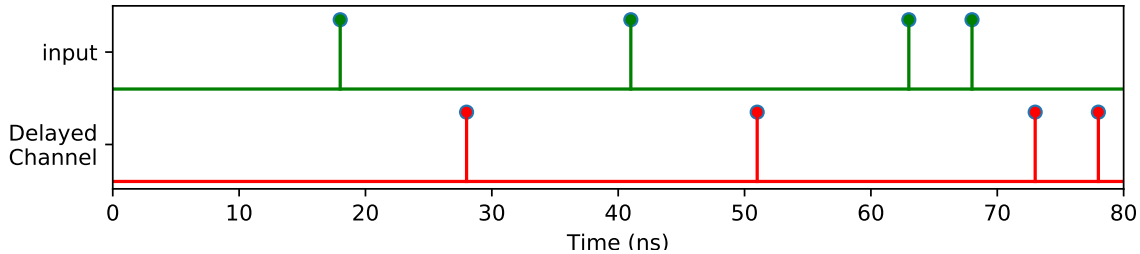
**class** `GatedChannel` (*tagger*, *input\_channel*, *gate\_start\_channel*, *gate\_stop\_channel*)

### Parameters

- **tagger** (`TimeTagger`) – time tagger object
- **input\_channel** (*int*) – channel which is gated
- **gate\_start\_channel** (*int*) – channel on which a signal detected will start the transmission of the `input_channel` through the gate

- **gate\_stop\_channel** (*int*) – channel on which a signal detected will stop the transmission of the input\_channel through the gate

### 7.5.8 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** (*int*) – channel to be delayed
- **delay** (*int*) – amount of time to delay in ps

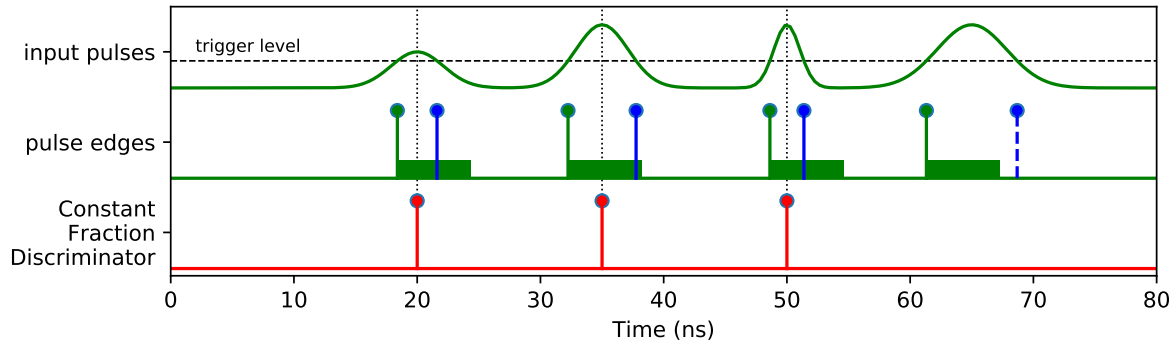
**setDelay** (*delay*)

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** (*int*) – Delay time in picoseconds

### 7.5.9 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.

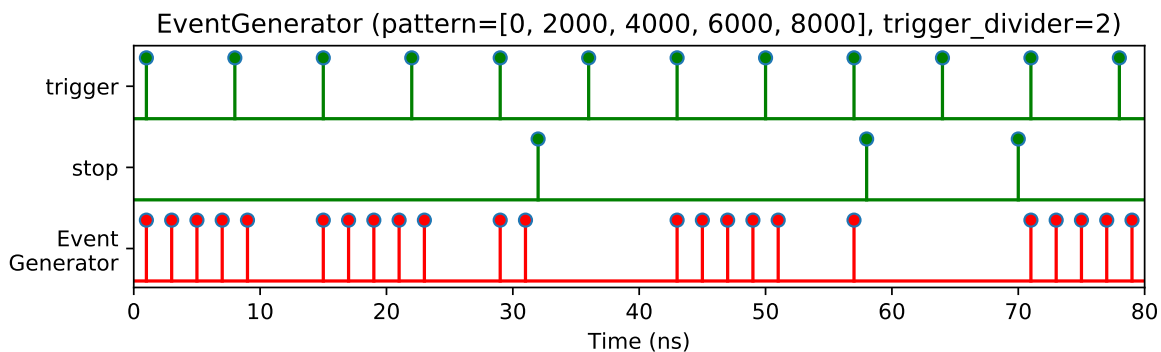
**Note:** The virtual CFD requires the time tags of the rising and falling edge. Hence, the transferred data of the input channel is twice the regular input rate.

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

#### Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **channels** (*list[int]*) – list of channels on which to perform CFD
- **search\_window** (*int*) – max pulse duration in picoseconds to be detected

### 7.5.10 EventGenerator





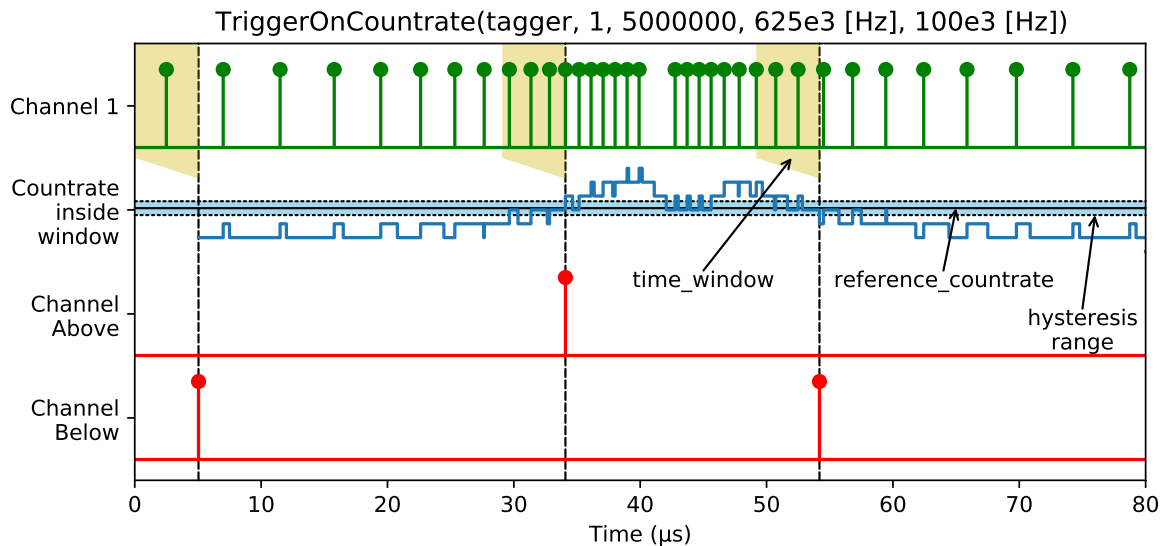
Emits an arbitrary pattern of timestamps for every trigger event. The number of trigger events can be reduced by *trigger\_divider*. The start of a new pattern does not abort the execution of unfinished patterns, so patterns may overlap. The execution of all running patterns can be aborted by a click 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** (*int*) – Channel number of the trigger signal.
- **pattern** (*list[int]*) – List of relative timestamps defining the pattern executed upon a trigger event.
- **trigger\_divider** (*int*) – Factor by which the number of trigger events is reduced. (default: 1)
- **divider\_offset** (*int*) – If *trigger\_divider* > 1, the *divider\_offset* the number of trigger clicks to be ignored before emitting the first pattern. (default: 0)
- **stop\_channel** (*int*) – Channel number of the stop channel. (optional)

### 7.5.11 TriggerOnCountRate



Measures the count rate inside a rolling time window and emits tags when a defined *reference\_countRate* is crossed. A *TriggerOnCountRate* object provides two virtual channels: The *above* channel is triggered when the count rate exceeds the threshold (transition from *below* to *above*). The *below* channel is triggered when the count rate falls below the threshold (transition from *above* to *below*).

To avoid the emission of multiple trigger tags in the transition area, the *hysteresis* count rate modifies the threshold with respect to the transition direction: An event in the *above* channel will be triggered when the channel is in the *below* state and rises to  $\text{reference\_countRate} + \text{hysteresis}$  or above. Vice versa, the *below* channel fires when the channel is in the *above* state and falls to the limit of  $\text{reference\_countRate} - \text{hysteresis}$  or below.

The time-tags are always injected at the end of the integration window. You can use the *DelayedChannel* to adjust the temporal position of the trigger tags with respect to the integration time window.

The very first tag of the virtual channel will be emitted *time\_window* after the instantiation of the object and will reflect the current state, so either *above* or *below*.

**class TriggerOnCountrate** (*tagger*, *input\_channel*, *time\_window*, *reference\_countrate*, *hysteresis*)

#### Parameters

- **tagger** (*TimeTagger*) – Time Tagger object instance.
- **input\_channel** (*int*) – Channel number of the channel whose count rate will control the trigger channels.
- **reference\_countrate** (*float*) – The reference count rate in Hz that separates the *above* range from the *below* range.
- **hysteresis** (*float*) – The threshold count rate in Hz for transitioning to the *above* threshold state is *countrate*  $\geq$  *reference\_countrate* + *hysteresis*, whereas it is *countrate*  $\leq$  *reference\_countrate* - *hysteresis* for transitioning to the *below* threshold state. The hysteresis avoids the emission of multiple trigger tags upon a single transition.
- **time\_window** (*int*) – Rolling time window size in ps. The count rate is analyzed within this time window and compared to the threshold count rate.

**getChannelAbove** ()

Get the channel number of the *above* channel.

**getChannelBelow** ()

Get the channel number of the *below* channel.

**getChannels** ()

Get both virtual channel numbers: [*getChannelAbove* (), *getChannelBelow* ()]

**getCurrentCountrate** ()

Get the current count rate averaged within the *time\_window*.

**injectCurrentState** ()

Emit a time-tag into the respective channel according to the current state. This is useful if you start a new measurement that requires the information. The function returns whether it was possible to inject the event. The injection is not possible if the Time Tagger is in overflow mode or the time window has not passed yet. The function call is non-blocking.

**isAbove** ()

Returns whether the Virtual Channel is currently in the *above* state.

**isBelow** ()

Returns whether the Virtual Channel is currently in the *below* state.

## 7.6 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. 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 a *.getData* () method.

## 7.6.1 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 spectroscopy.

**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. See *Raw Time-Tag-Stream access* to get an overview about the possibilities for the raw time-tag-stream access.

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

**FileReader** Allows you to read time-tags from a file written by the *FileWriter*.

## 7.6.2 Common methods

### **clear()**

Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.

### **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 return 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

### **waitUntilFinished(timeout)**

Blocks the execution until the measurement has finished. Can be used with *startFor()*.

**Parameters** **timeout** (*int*) – optional, default: -1. timeout in milliseconds. Negative means no timeout, zero returns immediately.

**Returns** True if the measurement has finished, false on timeout

**Return type** bool

### **getCaptureDuration()**

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

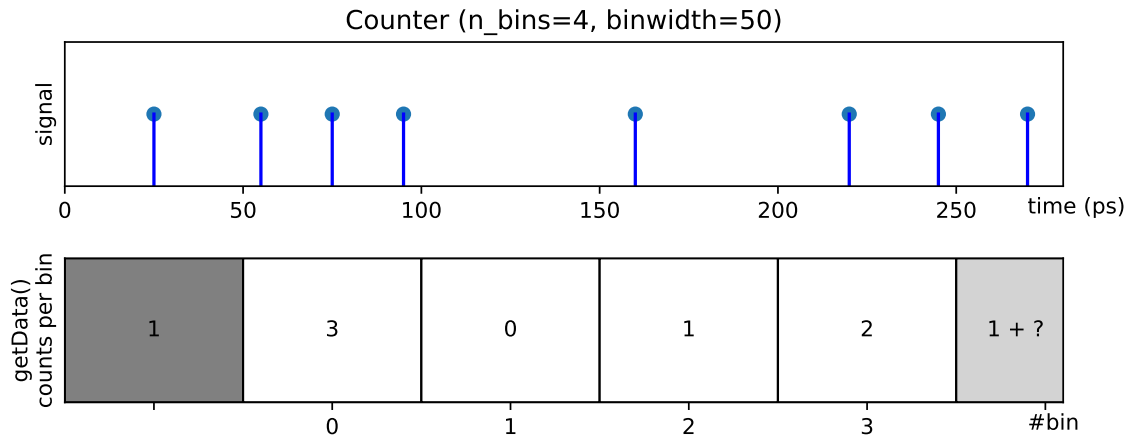
**Returns** Capture duration in ps

**Return type** int

## 7.6.3 Event counting



## 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[int]*) – channels used for counting tags
- **binwidth** (*int*) – bin width in ps
- **n\_values** (*int*) – number of bins (data buffer size)

*See all common methods*

**getData()**

**Returns** An array of size ‘number of channels’ by *n\_values* containing the current values of the circular buffer (counts in each bin).

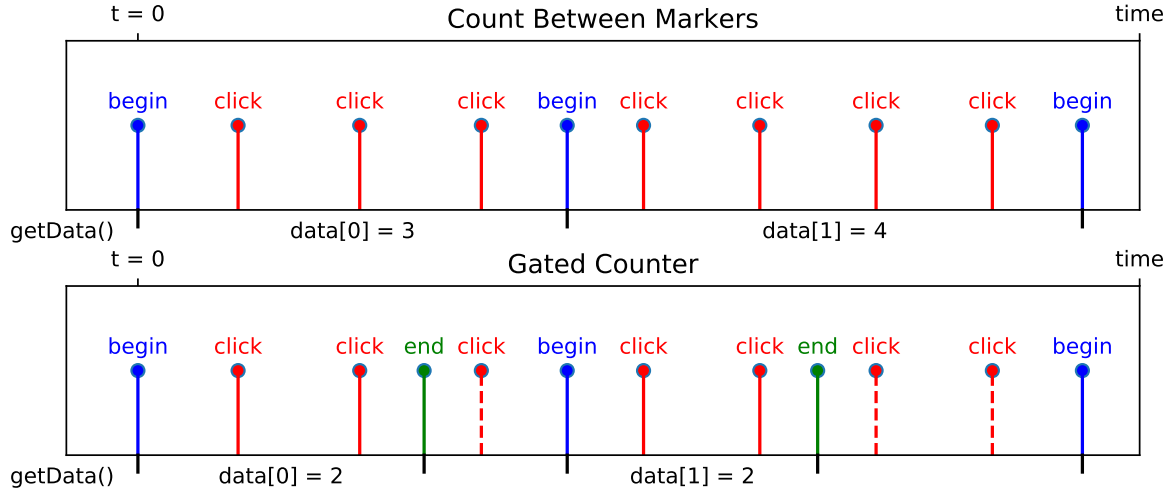
**Return type** 2D\_array[int]

**getIndex()**

**Returns** A vector of size *n\_values* containing the time bins in ps.

**Return type** 1D\_array[int]

## 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** (*int*) – channel on which clicks are received, gated by *begin\_channel* and *end\_channel*
- **begin\_channel** (*int*) – channel that triggers the beginning of counting and stepping to the next value
- **end\_channel** (*int*) – channel that triggers the end of counting
- **n\_values** (*int*) – number of values stored (data buffer size)

*See all common methods*

**getData()**

**Returns** Array of size  $n\_values$  containing the acquired counter values.

**Return type** 1D\_array[int]

**getIndex()**

**Returns** Vector of size  $n\_values$  containing the time in ps of each start click in respect to the very first start click.

**Return type** 1D\_array[int]

**getBinWidths()**

**Returns** Vector of size  $n\_values$  containing the time differences of ‘start -> (next start or stop)’ for the acquired counter values.

**Return type** 1D\_array[int]

**ready()**

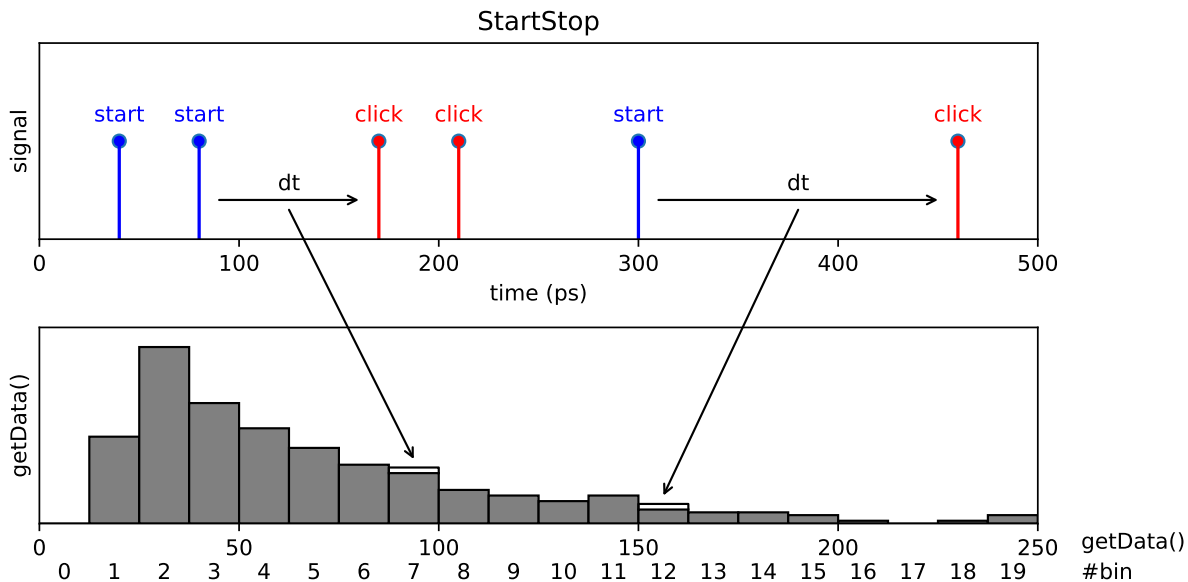
**Returns** True when the entire array is filled.

**Return type** bool

## 7.6.4 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** (*int*) – channel on which stop clicks are received
- **start\_channel** (*int*) – channel on which start clicks are received
- **binwidth** (*int*) – bin width in ps

*See all common methods*

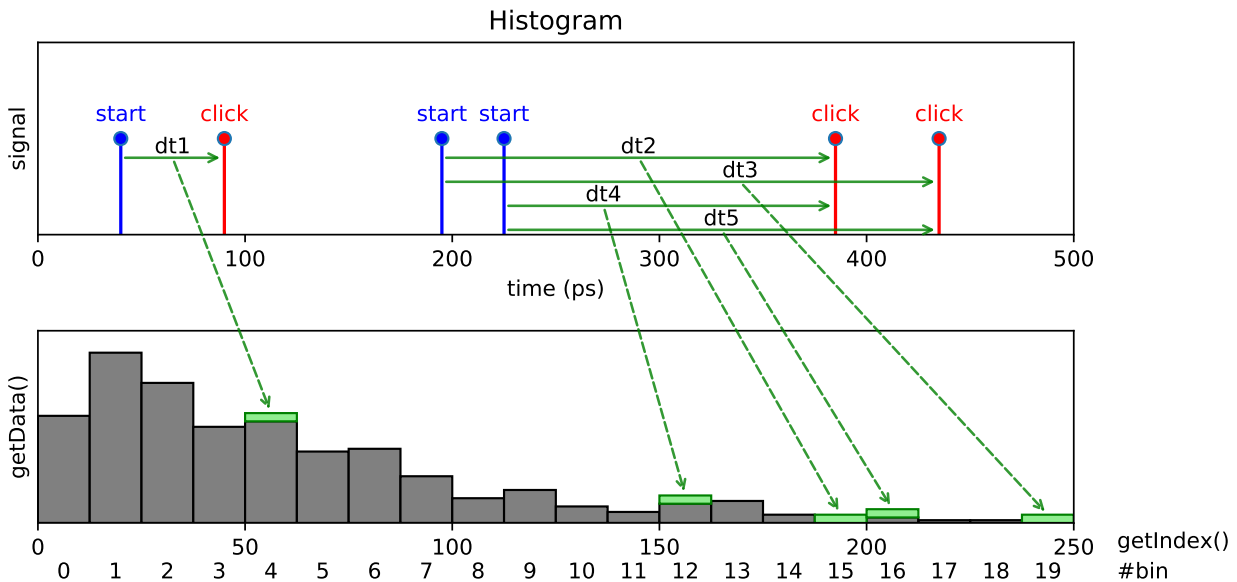
**getData()**



**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.

**Return type** 2D\_array[int]

## 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** (*int*) – channel on which clicks are received
- **start\_channel** (*int*) – channel on which start clicks are received
- **binwidth** (*int*) – bin width in ps
- **n\_bins** (*int*) – the number of bins in the histogram

*See all common methods*

**getData()**

**Returns** A one-dimensional array of size *n\_bins* containing the histogram.

**Return type** 1D\_array[int]

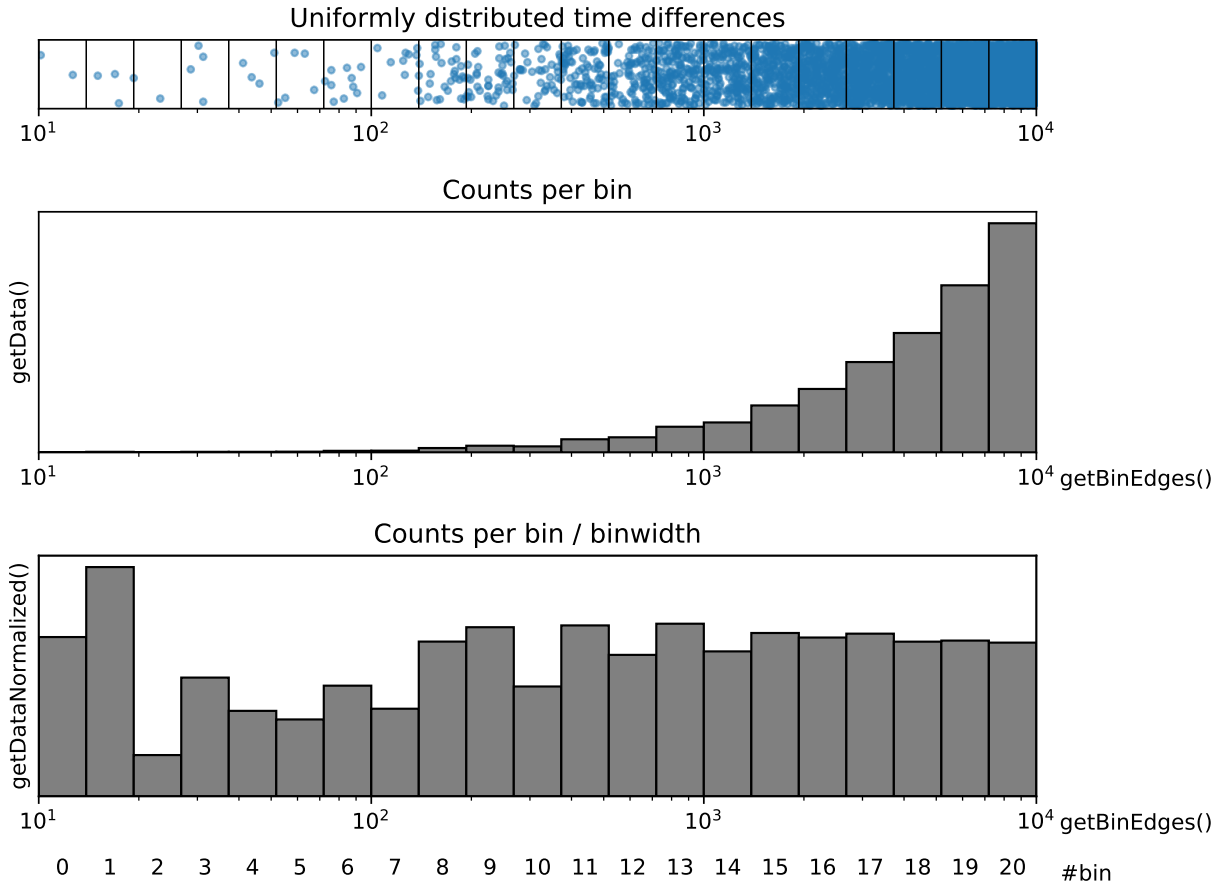
**getIndex()**

**Returns** A vector of size *n\_bins* containing the time bins in ps.

**Return type** 1D\_array[int]

## 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** (*int*) – channel on which clicks are received
- **start\_channel** (*int*) – channel on which start clicks are received
- **exp\_start** (*float*) – exponent  $10^{\text{exp\_start}}$  in seconds where the very first bin begins
- **exp\_stop** (*float*) – exponent  $10^{\text{exp\_stop}}$  in seconds where the very last bin ends
- **n\_bins** (*int*) – the number of bins in the histogram

---

**Note:** After initializing the measurement (or after an overflow) no data is accumulated in the histogram until the full histogram duration has passed to ensure a balanced count accumulation over the full histogram.

---

See all common methods

**getData ()**

**Returns** A one-dimensional array of size  $n\_bins$  containing the histogram.

**Return type** 1D\_array[int]

**getDataNormalizedCountsPerPs ()**

**Returns** The counts normalized by the binwidth of each bin.

**Return type** 1D\_array[float]

**getDataNormalizedG2 ()**

The counts normalized by the binwidth of each bin and the average count rate. This matches the implementation of [Correlation.getDataNormalized\(\)](#)

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

where  $\Delta t$  is the capture duration,  $N_1$  and  $N_2$  are number of events in each channel.

**Returns** The counts normalized by the binwidth of each bin and the average count rate.

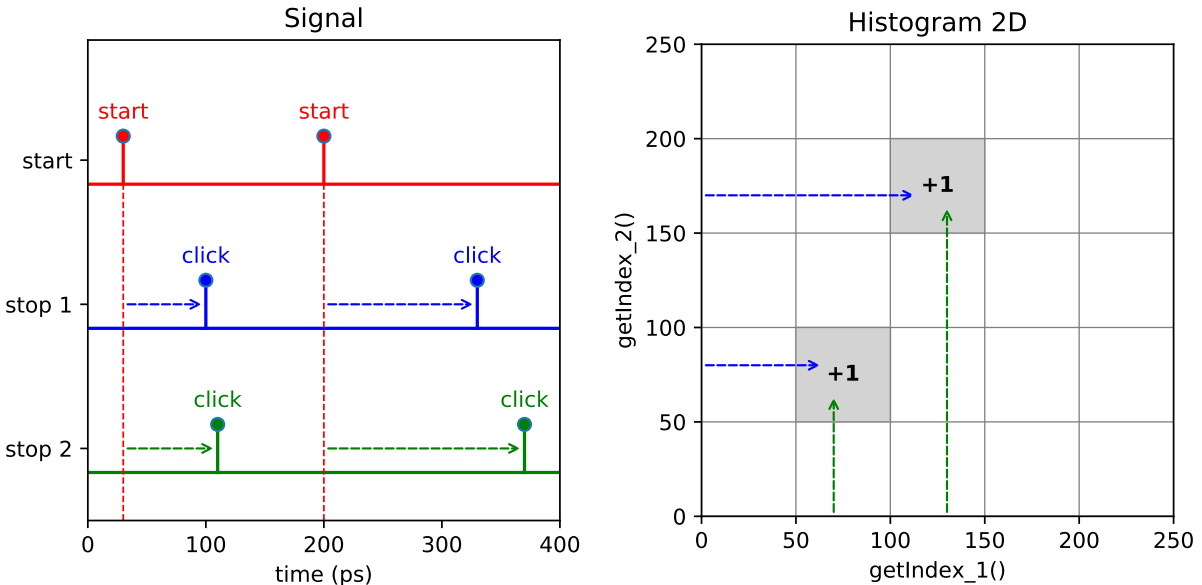
**Return type** 1D\_array[float]

**getBinEdges ()**

**Returns** A vector of size  $n\_bins+1$  containing the bin edges in picoseconds.

**Return type** 1D\_array[int]

## 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 acquired 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** (*int*) – channel on which start clicks are received
- **stop\_channel\_1** (*int*) – channel on which stop clicks for the time axis 1 are received
- **stop\_channel\_2** (*int*) – channel on which stop clicks for the time axis 2 are received
- **binwidth\_1** (*int*) – bin width in ps for the time axis 1
- **binwidth\_2** (*int*) – bin width in ps for the time axis 2
- **n\_bins\_1** (*int*) – the number of bins along the time axis 1
- **n\_bins\_2** (*int*) – the number of bins along the time axis 2

*See all common methods*

**getData()**

**Returns** A two-dimensional array of size *n\_bins\_1* by *n\_bins\_2* containing the 2D histogram.

**Return type** 2D\_array[int]

**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](#).

**Returns** A three-dimensional array of size *n\_bins\_1* x *n\_bins\_2* x 2

**Return type** 3D\_array[int]

**getIndex\_1()**

**Returns** A vector of size *n\_bins\_1* containing the bin locations in ps for the time axis 1.

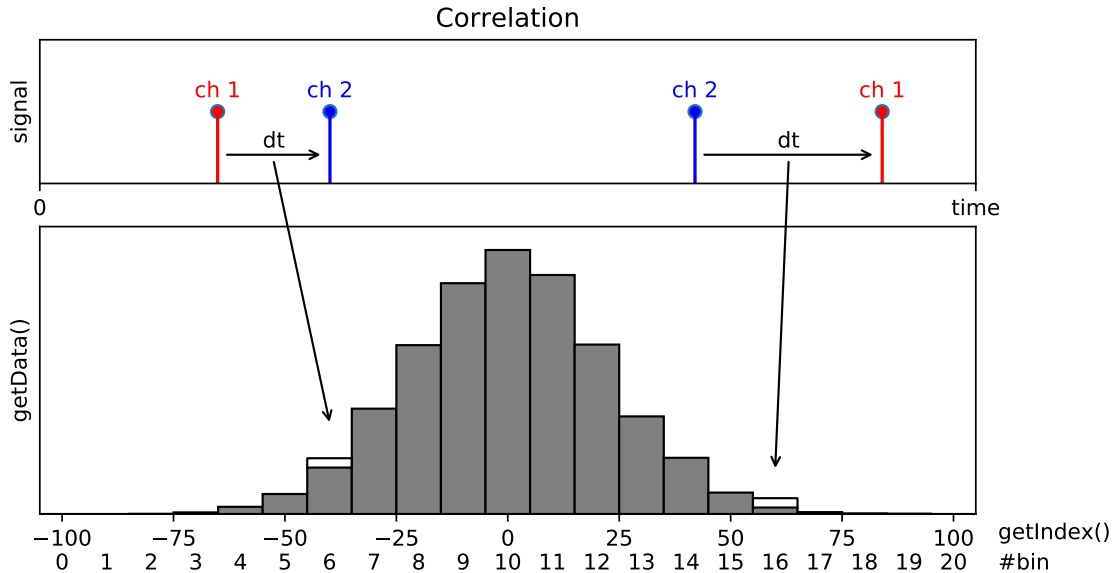
**Return type** 1D\_array[int]

**getIndex\_2()**

**Returns** A vector of size *n\_bins\_2* containing the bin locations in ps for the time axis 2.

**Return type** 1D\_array[int]

## 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** (*int*) – channel on which (stop) clicks are received
- **channel\_2** (*int*) – 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** (*int*) – bin width in ps
- **n\_bins** (*int*) – the number of bins in the resulting histogram

*See all common methods*

**getData()**

**Returns** A one-dimensional array of size *n\_bins* containing the histogram.

**Return type** 1D\_array[int]

**getDataNormalized()**

Return the data normalized as:

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

where  $\Delta t$  is the capture duration,  $N_1$  and  $N_2$  are number of events in each channel.

**Returns** Data normalized by the binwidth and the average count rate.

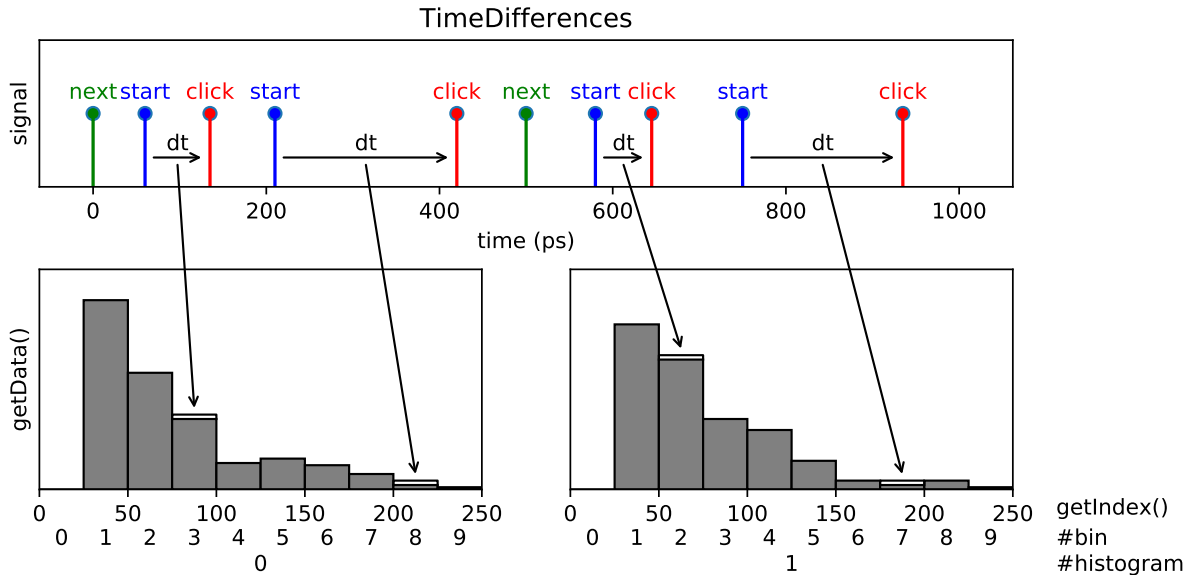
**Return type** 1D\_array[float]

**getIndex()**

**Returns** A vector of size  $n\_bins$  containing the time bins in ps.

**Return type** 1D\_array[int]

## 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** (`int`) – channel on which stop clicks are received
- **start\_channel** (`int`) – channel that sets start times relative to which clicks on the click channel are measured
- **next\_channel** (`int`) – channel that increments the histogram index
- **sync\_channel** (`int`) – channel that resets the histogram index to zero
- **binwidth** (`int`) – binwidth in picoseconds
- **n\_bins** (`int`) – number of bins in each histogram
- **n\_histograms** (`int`) – number of histograms

*See all common methods*

**getData()**

**Returns** A two-dimensional array of size *n\_bins* by *n\_histograms* containing the histograms.

**Return type** 2D\_array[int]

**getIndex()**

**Returns** A vector of size *n\_bins* containing the time bins in ps.

**Return type** 1D\_array[int]

**setMaxCounts()**

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

**getCounts()**

**Returns** The number of rollovers (histogram index resets).

**Return type** int

**ready()**

**Returns** True when the required number of rollovers set by `setMaxCounts()` has been reached.

**Return type** bool

**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





- **start\_channel** (*int*) – channel that sets start times relative to which clicks on the click channel are measured
- **next\_channels** (*list[int]*) – vector of channels that increments the histogram index
- **sync\_channels** (*list[int]*) – vector of channels that resets the histogram index to zero
- **n\_histograms** (*int*) – vector of numbers of histograms per dimension
- **binwidth** (*int*) – width of one histogram bin in ps
- **n\_bins** (*int*) – number of bins in each histogram

*See all common methods*

See methods of *TimeDifferences* class.

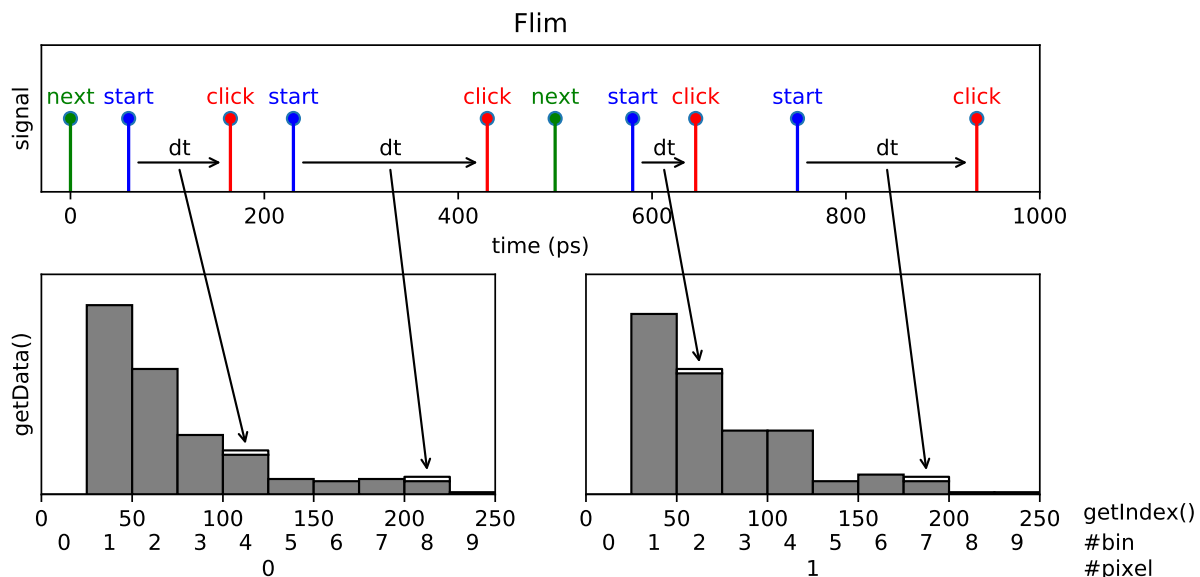
## 7.6.5 Fluorescence-lifetime imaging (FLIM)

This section describes the Flim related measurements classes of the Time Tagger API.

### Flim

Changed in version 2.7.2.

**Note:** The Flim (beta) implementation is a not final yet. It has a very advanced functionality, but details are subject to change. Please give us feedback ([support@swabianinstruments.com](mailto:support@swabianinstruments.com)) when you encounter issues or when you have ideas for additional functionality.



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.

There are two different classes to support the FLIM measurement: *Flim* and *FlimBase*. *Flim* provides a versatile high-level API. *FlimBase* instead provides the essential functionality with no overhead to perform Flim measurements. *FlimBase* is based on a callback approach.

Please visit the Python example folder for a reference implementation.

---

**Note:** Up to version 2.7.0, the FLIM implementation was very limited and has been rewritten completely with 2.7.2. You can use the following 1 to 1 replacement to get the old FLIM behavior:

---

```
# FLIM up to version 2.7.0:
Flim(tagger, click_channel=1, start_channel=2, next_channel=3,
     binwidth=100, n_bins=1000, n_pixels=320*240)

# FLIM 2.7.0 replacement 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)
```

```
class Flim(tagger, start_channel, click_channel, pixel_begin_channel, n_pixels, n_bins, binwidth[,
           pixel_end_channel=CHANNEL_UNUSED, frame_begin_channel=CHANNEL_UNUSED, finish_after_outputframe=0, n_frame_average=1, pre_initialize=True])
```

High-Level class for implementing FLIM measurements. The Flim class includes buffering of images and several analysis methods.

The methods are split into different groups.

The `getCurrent...` references the active frame.

The `getReady...` references the last full frame acquired.

The `getSummed...` operates on all frames which have been captured so far including or excluding the current active frame via the optional parameter `only_ready_frames`.

The `get...Ex` returns instead of an array, a *FlimFrameInfo* which contains more information than only the raw array.

The class provides an `frameReady()` callback, which can be used to analyze the data when a frame is completed.

### Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **start\_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click\_channel** (*int*) – start\_channel channel on which start clicks are received for the time differences histogramming
- **pixel\_begin\_channel** (*int*) – start of a pixel (histogram)
- **n\_pixels** (*int*) – number of pixels (histograms) of one frame

- **n\_bins** (*int*) – number of histogram bins for each pixel
- **binwidth** (*int*) – bin size in picoseconds
- **pixel\_end\_channel** (*int*) – end marker of a pixel - incoming clicks on the click\_channel will be ignored afterward. (optional)
- **frame\_begin\_channel** (*int*) – start the frame, or reset the pixel index. (optional)
- **finish\_after\_outputframe** (*int*) – sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0, one frame is stored and the measurement runs continuously. (optional, default: 0)
- **n\_frame\_average** (*int*) – average multiple input frames into one output frame, (optional, default: 1)
- **pre\_initialize** (*bool*) – initializes the measurement on constructing. (optional)

*See all common methods*

**getCurrentFrame()**

**Returns** The histograms for all pixels of the currently active frame.

**Return type** 3D\_array[int]

**getCurrentFrameEx()**

**Returns** The currently active frame.

**Return type** *FlimFrameInfo*

**getCurrentFrameIntensity()**

**Returns** The intensities of all pixels of the currently active frame. The pixel intensity is defined by the number of counts acquired within the pixel divided by the respective integration time.

**Return type** 2D\_array[float]

**getFramesAcquired()**

**Returns** The number of frames that have been completed so far, since the creation / last clear of the object.

**Return type** int

**getIndex()**

**Returns** A vector of size n\_bins containing the time bins in ps.

**Return type** 1D\_array[int]

**getReadyFrame([index=-1])**

**Parameters** **index** (*int*) – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional)

**Returns** The histograms for all pixels according to the frame index given. If the index is -1, it will return the last frame, which has been completed. When stop\_after\_outputframe is 0, the index value must be -1. If index >= stop\_after\_outputframe, it will throw an error.

**Return type** 3D\_array[int]

**getReadyFrameEx([index=-1])**

**Parameters** **index** (*int*) – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional)

**Returns** The frame according to the index given. If the index is -1, it will return the last completed frame. When `stop_after_outputframe` is 0, index must be -1. If `index >= stop_after_outputframe`, it will throw an error.

**Return type** *FlimFrameInfo*

**getReadyFrameIntensity** (`[index=-1]`)

**Parameters** **index** (*int*) – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional)

**Returns** The intensities according to the frame index given. If the index is -1, it will return the intensity of the last frame, which has been completed. When `stop_after_outputframe` is 0, the index value must be -1. If `index >= stop_after_outputframe`, it will throw an error. The pixel intensity is defined by the number of counts acquired within the pixel divided by the respective integration time.

**Return type** 2D\_array[float]

**getSummedFrames** (`[only_ready_frames=True, clear_summed=False]`)

**Parameters**

- **only\_ready\_frames** – If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional)
- **clear\_summed** – If true, the summed frames memory will be cleared. (optional)

**Returns** The histograms for all pixels. The counts within the histograms are integrated since the start or the last clear of the measurement.

**Return type** 2D\_array[int]

**getSummedFramesEx** (`[only_ready_frames=True, clear_summed=False]`)

**Parameters**

- **only\_ready\_frames** (*bool*) – If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional)
- **clear\_summed** (*bool*) – If true, the summed frames memory will be cleared. (optional)

**Returns** A *FlimFrameInfo* that represents the sum of all acquired frames.

**Return type** *FlimFrameInfo*

**getSummedFramesIntensity** (`[only_ready_frames=True, clear_summed=False]`)

**Parameters**

- **only\_ready\_frames** (*bool*) – If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional)
- **clear\_summed** (*bool*) – If true, the summed frames memory will be cleared. (optional)

**Returns** The intensities of all pixels summed over all acquired frames. The pixel intensity is the number of counts within the pixel divided by the integration time.

**Return type** 2D\_array[float]

**isAcquiring** ()

**Returns** A boolean which tells the user if the class is still acquiring data. It can only reach the false state for `stop_after_outputframe > 0`. This should differ from `isRunning()` as once rendering is done, it can't be started again.

**Return type** bool

**frameReady** (*frame\_number*, *data*, *pixel\_begin\_times*, *pixel\_end\_times*, *frame\_begin\_time*, *frame\_end\_time*)

**Parameters**

- **frame\_number** (*int*) – current Frame number
- **data** (*list[int]*) – raw histogram data for all pixels
- **pixel\_begin\_times** (*list[int]*) – start time for each pixel
- **pixel\_end\_times** (*list[int]*) – end time for each pixel
- **frame\_begin\_time** (*int*) – start time of the frame
- **frame\_end\_time** (*int*) – end time of the frame

The method is called when a frame is completed. Compared to `on_frame_end()`, it provides various related data when invoked.

**on\_frame\_end()**

Virtual function which can be overwritten in C++. The method is called when a frame is completed.

## FlimFrameInfo

This is a simple class that contains FLIM frame data and provides convenience accessor methods.

---

**Note:** Objects of this class are returned by the methods of the FLIM classes. Normally user will not construct `FlimFrameInfo` objects themselves.

---

**class FlimFrameInfo**

**pixels**

number of pixels of the frame

**bins**

number of bins of each histogram

**frame\_number**

current frame number

**pixel\_count**

current pixel position

**getFrameNumber()**

**Returns** The frame number, starting from 0 for the very first frame acquired. If the index is -1, it is an invalid frame which is returned on error.

**Return type** int

**isValid()**

**Returns** A boolean which tells if this frame is valid or not. Invalid frames are possible on errors, such as asking for the last completed frame when no frame has been completed so far.

**Return type** bool

**getPixelPosition()**

**Returns** A value which tells how many pixels were processed for this frame.

**Return type** int

**getHistograms()**

**Returns** All histograms of the frame.

**Return type** 3D\_array[int]

**getIntensities()**

**Returns** The summed counts of each histogram divided by the integration time.

**Return type** 2D\_array[float]

**getSummedCounts()**

**Returns** The summed counts of each histogram.

**Return type** 2D\_array[int]

**getPixelBegins()**

**Returns** An array of the start timestamps of each pixel.

**Return type** 2D\_array[int]

**getPixelEnds()**

**Returns** An array of the end timestamps of each pixel.

**Return type** 2D\_array[int]

## FlimBase

The *FlimBase* provides only the most essential functionality for FLIM tasks. The benefit from the reduced functionality is that it is very memory and CPU efficient. The class provides the *frameReady()* callback, which must be used to analyze the data.

```
class FlimBase (tagger,      start_channel,      click_channel,      pixel_begin_channel,      n_pixels,
                n_bins,      binwidth[,      pixel_end_channel=CHANNEL_UNUSED,
                frame_begin_channel=CHANNEL_UNUSED,      finish_after_outputframe=0,
                n_frame_average=1, pre_initialize=True])
```

### Parameters

- **tagger** (*TimeTagger*) – time tagger object instance
- **start\_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click\_channel** (*int*) – start\_channel channel on which start clicks are received for the time differences histogramming
- **pixel\_begin\_channel** (*int*) – start of a pixel (histogram)
- **n\_pixels** (*int*) – number of pixels (histograms) of one frame
- **n\_bins** (*int*) – number of histogram bins for each pixel
- **binwidth** (*int*) – bin size in picoseconds

- **pixel\_end\_channel** (*int*) – end marker of a pixel - incoming clicks on the click\_channel will be ignored afterward. (optional, default: `CHANNEL_UNUSED`)
- **frame\_begin\_channel** (*int*) – start the frame, or reset the pixel index. (optional, default: `CHANNEL_UNUSED`)
- **finish\_after\_outputframe** (*int*) – sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0, one frame is stored and the measurement runs continuously. (optional, default: 0)
- **n\_frame\_average** (*int*) – average multiple input frames into one output frame. (optional, default: 1)
- **pre\_initialize** (*bool*) – initializes the measurement on constructing. (optional, default: True)

*See all common methods*

**isAcquiring** ()

**Returns** A boolean which tells the user if the class is still acquiring data. It can only reach the false state for stop\_after\_outputframe > 0. This should differ from `isRunning()` as once rendering is done, it can't be started again.

**Return type** bool

**frameReady** (*frame\_number*, *data*, *pixel\_begin\_times*, *pixel\_end\_times*, *frame\_begin\_time*, *frame\_end\_time*)

**Parameters**

- **frame\_number** (*int*) – current Frame number
- **data** (*2D\_array[int]*) – 2D array containing the raw histogram data for all pixels
- **pixel\_begin\_times** (*list[int]*) – start time for each pixel
- **pixel\_end\_times** (*list[int]*) – end time for each pixel
- **frame\_begin\_time** (*int*) – start time of the frame
- **frame\_end\_time** (*int*) – end time of the frame

The method is called when a frame is completed. Compared to `on_frame_end()`, it provides various related data when invoked.

**on\_frame\_end** ()

Virtual function which can be overwritten in C++. The method is called when a frame is completed.

## 7.6.6 Time-tag-streaming

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

## Time tag format

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

Size	Type	Description
8 bit	enum <i>OverflowType</i>	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

### **class 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 number of lost events per channel within an overflow interval. Might be sent repeatedly for larger number 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*, *n\_max\_events*, *channels*)

### **Parameters**

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

*See all common methods*

### **getData()**

**Returns** 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 time-tag only once, also with no data loss.

**Return type** *TimeTagStreamBuffer*



**class TimeTagStreamBuffer**

**getTimestamps ()**  
Returns an array of timestamps.

**Returns** Event timestamps in picoseconds for all chosen channels.

**Return type** list[int]

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

**Returns** Channel number for each detected event.

**Return type** list[int]

**getOverflows ()**  
Deprecated since version 2.5.0.

Returns an array of overflow flags for every timestamp.

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

**Returns** Event type value for each detected event.

**Return type** 1D\_array(*OverflowType*)

**getMissedEvents ()**  
Returns an array of missed event counts during an overflow situation.

**Returns** Missed events value for each detected event.

**Return type** 1D\_array[int]

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

**Returns** True/False

**Return type** bool

**FileWriter**

Writes the time-tag-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** (*str*) – name of the file to store to
- **channels** (*list[int]*) – 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()*.

---

*See all common methods*

**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* (*str*) – 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()*.

**Return type** int

**getTotalEvents** ()

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

**Return type** int

**getTotalSize** ()

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

**Return type** int

## FileReader

This measurement allows you to read data files store with *FileWriter*. 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 constructor. The *FileReader* automatically continues to read files that were split by the *FileWriter*.

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

### **getData** (*size\_t n\_events*)

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 exactly 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()*.

**Parameters** *n\_events* (*int*) – Number of timetags to read from the file.

**Returns** A buffer of size *n\_events*.

**Return type** *TimeTagStreamBuffer*

### **hasData** ()

**Returns** *True* if more data is available for reading, *False* if all data has been read from all the files specified in the class constructor.

**Return type** *bool*

### **getConfiguration** ()

**Returns** A JSON formatted string that contains the Time Tagger configuration at the time of file creation.

**Return type** str

## 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** (*str*) – name of the file to dump to
- **max\_tags** (*int*) – stop after this number of tags has been dumped. Negative values will dump forever
- **channels** (*list[int]*) – list of real or virtual channels which are dumped to the file (when empty or not passed all active channels are dumped)

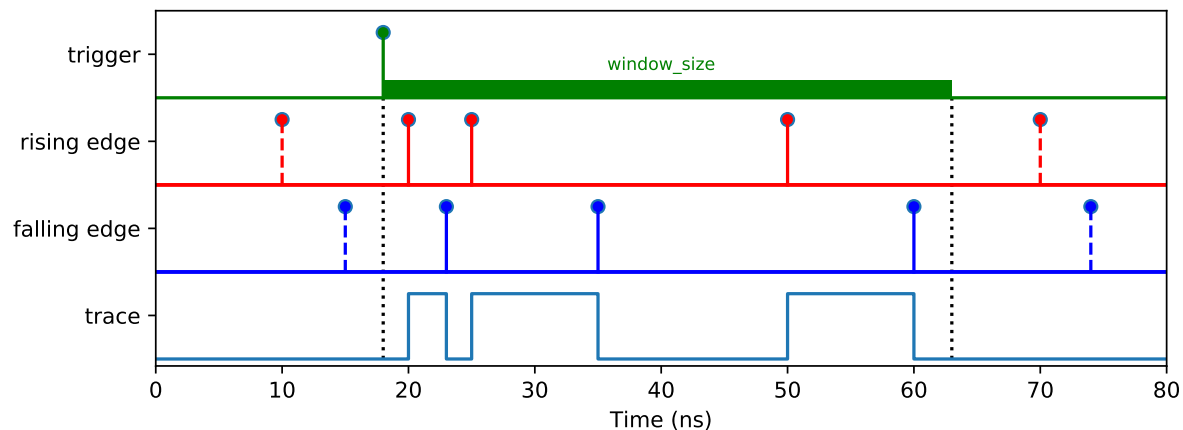
**clear** ()

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[int]*) – List of channels
- **trigger\_channel** (*int*) – Channel number of the trigger signal
- **window\_size** (*int*) – Time window in picoseconds
- **n\_traces** (*int*) – Number of trigger events to be detected
- **n\_max\_events** (*int*) – Max number of events to be detected

*See all common methods*

**getData** ()

Returns a tuple of the size equal to the number of *event\_channels*, where each element is a tuple of *Event*.

**Returns** Event list for each channel.

**Return type** tuple[tuple[*Event*]]

**class Event**

Pair of the timestamp and the new state.

**time**

**Type** int

**state**

**Type** *State*

**class State**

Current input state. Can be unknown because no edge has been detected on the given channel after initialization or an overflow.

**UNKNOWN**

**HIGH**

**LOW**

## 7.6.7 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. You can pass a Time Tagger proxy-object returned by *SynchronizedMeasurements.getTagger()* to every measurement you create. This will simultaneously disable their autostart and register for synchronization.

**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.

---

**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 *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 synchronized behavior.

---

### 7.6.8 Custom Measurements

The class `CustomMeasurement` allows you to access the raw time tag stream with very little overhead. By inheriting from `CustomMeasurement`, you can implement your fully customized measurement class. The `CustomMeasurement.process()` method of this class will be invoked as soon as new data is available. Note that this functionality is only available for C++, C# and Python. You find examples of how to use the `CustomMeasurement` in your examples folder.

**class** `CustomMeasurement` (*tagger*)

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

The constructor of the `CustomMeasurement` class itself takes only the parameter *tagger*. When you sub-class your own measurement, you can add to your constructor any parameters that are necessary for your measurement. You can find detailed examples in your example folder.

**process** (*incoming\_tags*, *begin\_time*, *end\_time*)

**Parameters**

- **incoming\_tags** – `Tag[][struct{type, missed_events, channel, time}]`, the chunk of time-tags to be processed in this call of `process()`. This is an external reference that is shared with other measurements and might be overwritten for the next call. So if you need to store tags, create a copy.
- **begin\_time** (*int*) – The begin time of the data chunk.
- **end\_time** (*int*) – The end time of the data chunk

Override the `process()` method to include your data processing. The method will be called by the Time Tagger backend when a new chunk of time-tags is available. You are free to execute any code you like, but be aware that this is the critical part when it comes to performance. In Python, it is advisable to use `Numpy arrays` for calculation or even pre-compiled code with `Numba` if an explicit iteration of the tags is necessary. Check the examples in your examples folder carefully on how to design the `process()` method.

**Note:** In Python, the *incoming\_tags* are a `structured Numpy array`. You can access single tags as well as arrays of tag entries directly:

```
first_tag = incoming_tags[0]
all_timestamps = incoming_tags['time']
```

*See all common methods*





## IN DEPTH GUIDES

This section contains articles that provide in depth details on the Time Tagger hardware and software.

### 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])
```

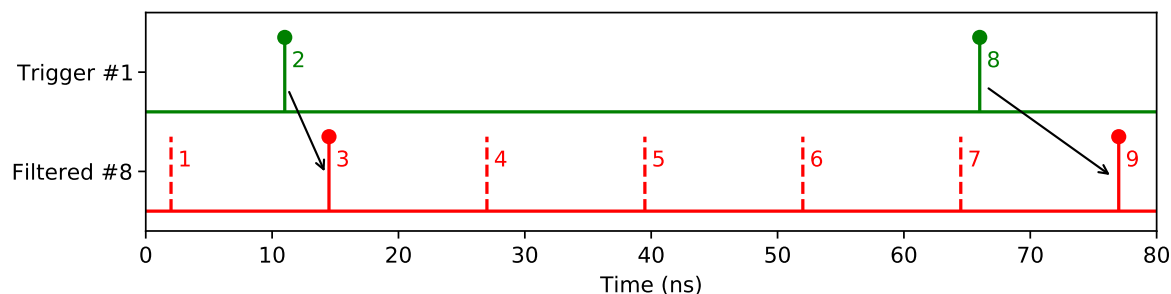
The details will be explained in the *Setup of the Conditional Filter* section.

#### 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])
```

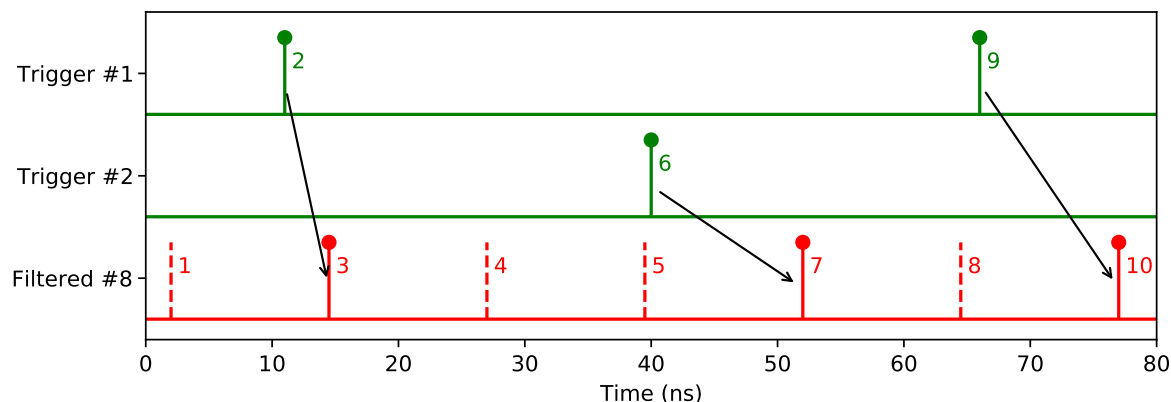


The Conditional Filter discards by default all signals of the filtered-channel. Only the very next event is transmitted after an event on the trigger-channel. In the example, click 2 opens the gate for click 3. When click 3 passes, it closes the gate and the subsequent events will be discarded until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

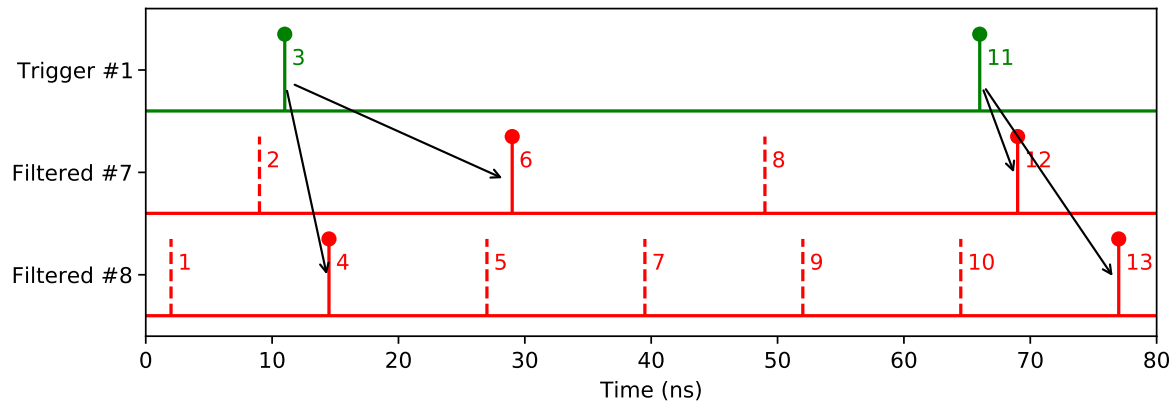


This is the typical use case when you detect photons with multiple detectors and want to correlate both with the common excitation laser.

### Multiple filtered channels

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

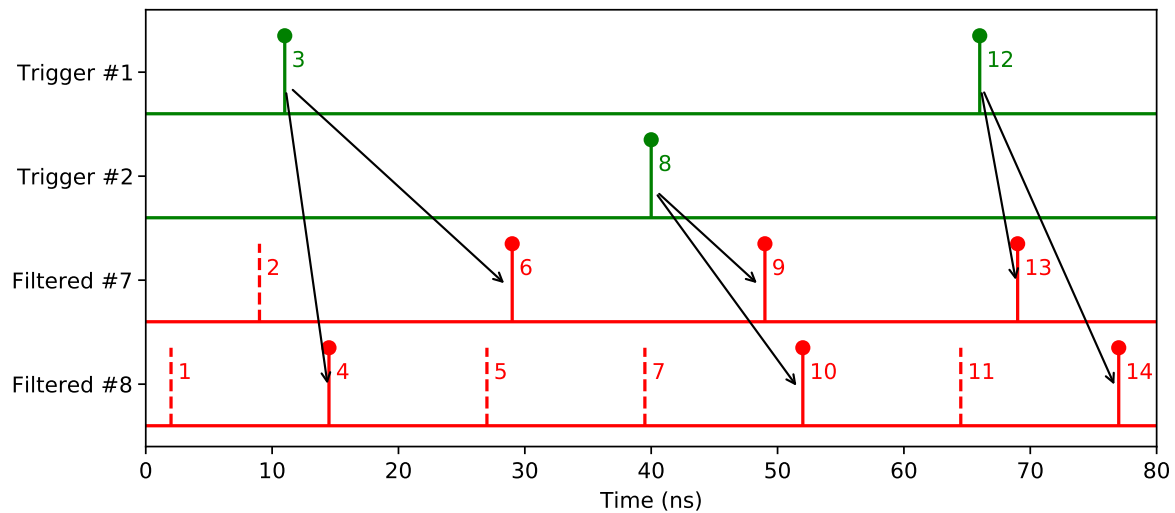
```
tagger.setConditionalFilter(trigger=[1], filtered=[7, 8])
```



### 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 is calibrated to compensate for this delay. This compensation is done on the device in case of the Time Tagger Ultra and in software for the Time Tagger 20 (see details below). 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

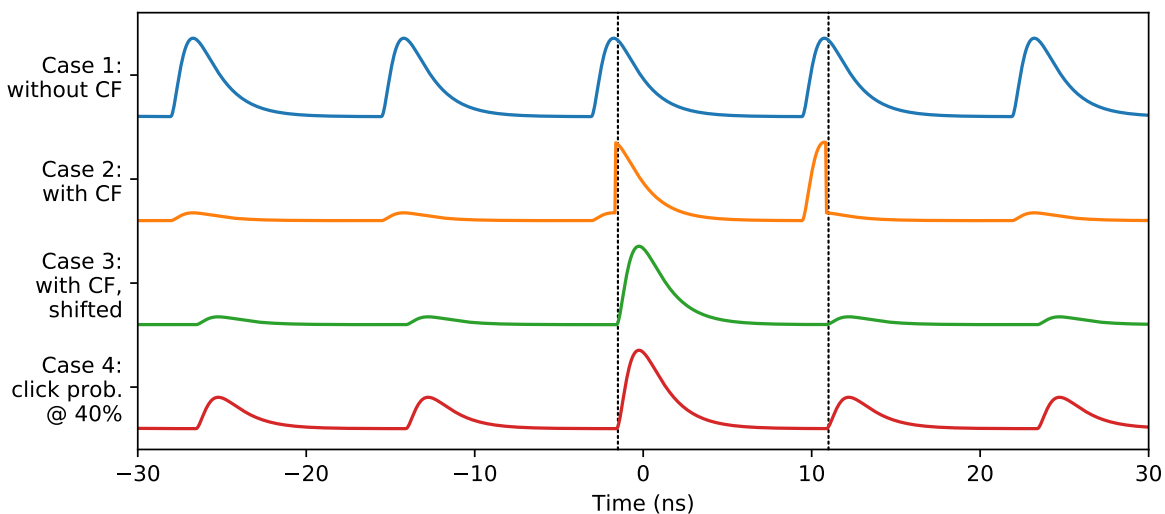
1. **Pulse enters the Time Tagger:** Up to the input connector, the user is in charge of the *external delays*. They can be controlled by changing cable lengths or optical pathways. The time tag generated by the Time Tagger should therefore represent the temporal order at the input connectors. This is 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 *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. It can be compensated in one of the later stages.
3. **Adjustable hardware delay (TT Ultra only!):** From software version 2.8.0 on, the Time Tagger Ultra is able to buffer and reorder the tags before the Conditional Filter. You can set an individual delay for every input stage by `TimeTagger.setDelayHardware()`. This behaves like an adjustable hardware delay and is calibrated by default to compensate for the physical hardware delay. It changes the behavior of the Conditional Filter tremendously, as you will see in the next stages.
4. **Conditional Filter:** As a first filter stage, the Conditional Filter is applied. The time tags of trigger channels and filtered channels are compared. If your device is able to introduce *Adjustable hardware delay*, this happens based on the timestamp including the *Hardware delay* compensation and the additional delay set by `TimeTagger.setDelayHardware()`. Otherwise, the raw *TDC time stamp* is used. In both cases, the time order of these stamps can deviate from the order of the *input time stamps* that you are dealing with usually.
5. **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.
6. **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.

7. **setDelaySoftware:** From now on, the Time Tagger hardware is not involved anymore. If your device does not provide an adjustable hardware delay, the software compensates now 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*). In any case, you can modify this compensation by `TimeTagger.setDelaySoftware()`.
8. **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 counterintuitive 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)
```

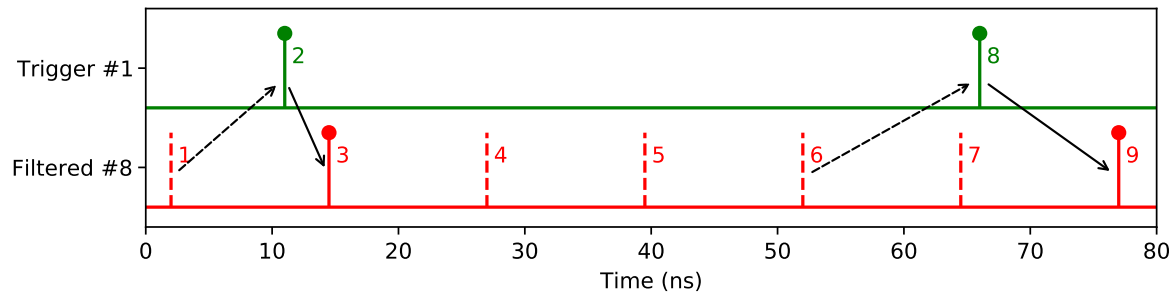


**Case 1:** Without the Conditional Filter set up, the Correlation measurement class provides a periodic signal. The periodicity is a result of the multi-start/multi-stop approach of the Correlation measurement: A click on the detector will contribute together with any laser synchronization pulse to 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, the strategy depends on your device: With a Time Tagger Ultra and software version 2.8.0 and later, you can use `TimeTagger.setDelayHardware()` to align your signals. In the case of a Time Tagger 20, however, you need to adjust your *external delays*. You might either modify optical path lengths or use cables of different lengths.

**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* (plus *adjustable onboard delays*, if available). 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 `TimeTagger.setConditionalFilter()` method expects two arguments, *trigger* and *filtered*, and accepts the optional boolean argument *hardwareDelayCompensation*:

```
tagger.setConditionalFilter(trigger: list[int],
                           filtered: list[int],
                           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*. If your device supports *adjustable onboard delays*, you should never set this value to False and you can ignore this section.

### 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 an empty lists or use the *TimeTagger.clearConditionalFilter()* method:

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

## 8.2 Raw Time-Tag-Stream access

There are several ways to access the raw time tags with the Time Tagger API. They can be split into two categories: dumping together with post-processing and on-the-fly processing. Both ways will be explained in the following. They are not exclusive so that you can combine them, also, with other measurements from our API in parallel.

## 8.2.1 Dumping and post-processing

All incoming time tags or selected channels of the Time Tagger can be stored on the hard drive via the *FileWriter*. Please visit the documentation and the provided programming examples of *FileWriter* for further details.

There are two ways for post-processing the dumped data:

### File Reader

By reading in the stored time tags with the *FileReader*, the tags stored can be processed natively in your preferred programming language. You find examples of how to use the *FileReader* in your examples folder.

### Virtual Time Tagger

The second option to process stored time tags is the *Time Tagger Virtual*. The *Time Tagger Virtual* allows you to use the full Time Tagger API to post-process your data. You find examples of how to use the *Time Tagger Virtual* in your examples folder.

## 8.2.2 On-the-fly processing

There are two options to process raw incoming data, the *TimeTagStream* and the *CustomMeasurement*, which will be explained in the following:

### TimeTagStream - high-level, lower performance

The *TimeTagStream* buffers the incoming raw data for on-the-fly processing. The *TimeTagStream* buffer must be polled to retrieve the tags. You find examples of how to use the *TimeTagStream* in your examples folder.

### CustomMeasurement - low-level, higher performance

The *CustomMeasurement* functionality allows you to access the raw time tag stream with very little overhead. By inheriting from *CustomMeasurement*, you can implement your fully customized measurement class. The *CustomMeasurement.process()* method of this class will be invoked as soon as new data is available. Note that this functionality is only available for C++, C#, and Python. You find examples of how to use the *CustomMeasurement* in your examples folder.

### CustomVirtualChannel - modify the time tag stream - C++ only

It is now possible for you to modify the time tag stream, like our API does by inserting time tags, e.g., via *Coincidence* or *DelayedChannel*. If you want to use this functionality, please contact Swabian Instruments support.



## IteratorBase - C++ only

### **class** IteratorBase

All measurements and virtual channels are derived from the *IteratorBase* class. You can see how to access the time tag stream on the deepest level with the provided C++ examples.

## 8.3 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 *TimeTagger.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.



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
```



## 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  $\geq 10$  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.

## REVISION HISTORY

### 11.1 V2.8.4 - 04.05.2021

- Fixed the initialization for a Virtual TimeTagger in the Web Application

### 11.2 V2.8.2 - 26.04.2021

- Fixed non appearing option to initialize in High-Res mode after upgrading/flashing the device in the Web Application.

### 11.3 V2.8.0 - 29.03.2021

#### Highlights

- High-resolution options for the Time Tagger Ultra series with a timing jitter of down to 4 ps RMS per channel.
- Hardware input delay on the Time Tagger Ultra series with picoseconds accuracy before the conditional filter.
- Reduced CPU load for Time Tagger Ultra

#### New Time Tagger Ultra features

- Reduced crosstalk and thermal drift on all channels.
- The Time Tagger hardware sound module can be activated and set via `TimeTagger.setSoundFrequency()`. It can be used, e.g., for optical alignment purposes (count rate -> frequency).

#### Changes

- Split `TimeTagger.setInputDelay()` into `TimeTagger.setDelayHardware()` and `TimeTagger.setDelaySoftware()`.
- `TimeTagger.getChannelList()` filter enum renamed to `ChannelEdge`.
- `TimeTagger.setNormalization()` can now be configured per channel.
- Changed the default port of the WebApp to 50120 to avoid collision with Jupiter Notebooks.

#### Features

- Added a `TriggerOnCountRate` virtual channel that generates events when a count rate crosses the given threshold value.
- Added support for Python 3.9.

- `waitUntilFinished()` and `sync` have an optional timeout parameter.

### Examples

- Mathematica: Added example for `FileWriter` and `TimeTaggerVirtual()`.
- LabVIEW: Fixed broken example (#14) and added it to the LabVIEW project.
- C++: Added an example for Custom Virtual Channel.

### Bug fixes

- Histogram can be used with `waitUntilFinished()` and `SynchronizedMeasurements`. `Histogram` is now derived from `IteratorBase()`.
- Displaying the singleton warning of `createTimeTagger` just once.
- Fixed string conversion issue for old Matlab versions.
- Hide “unused argument” warnings in the TimeTagger C++ headers.
- Fixed a bug for old Time Tagger Ultras, where the Web Application could not apply the license upgrade.

## 11.4 V2.7.6 - 26.04.2021

- Fixed `RuntimeError` “Got the USB error ‘UnsupportedFeature’” when calling `createTimeTagger()`

## 11.5 V2.7.4 - 19.04.2021

- Fixed a bug for old Time Tagger Ultras, where the Web Application could not apply the license upgrade.

## 11.6 V2.7.2 - 22.12.2020

### Highlights

- Reworked `Flim` implementation. Versatile high-level functionality with `Flim` and low-level CPU- and memory-efficient access via `FlimBase` and callbacks.
- Highly improved `TimeTaggerVirtual` performance taking use of multithreading.
- Support for direct time tag stream access via `Custom Measurements` in C# and Python - see examples in the installation folder.

### Improvements

- Added AnyCPU targeted .NET Assembly for C# wrappers. Available in GAC\_MSIL and the installation folder.
- More detailed error handling and human-readable error messages.
- Added `Conditional Filter` for `TimeTaggerVirtual`.
- Removed Intel’s `libmmd.dll` library dependency.
- All measurements have the new common method `waitUntilFinished()`, which can be used with `startFor()`.
- Warnings are printed with time information.
- Cleanup of the C++ measurements’ header file.



- Remote license upgrades can be performed via the web application.
- Reworked Python and C# examples.

#### Fixes

- *Countrate* no longer clears total counts on *start()*.
- Implemented *TimeTagger.getChannelList()* and *TimeTagger.waitForFence()* in Matlab.
- Fixed *TimeTagger.setDeadtime()* for the *TimeTaggerVirtual* using *TimeTagger.setTestSignal()*.
- Fixed a frequent crash in *FileWriter* with high data rates and multiple files.
- Fixed a crash in deleting measurements still registered to *SynchronizedMeasurements*.
- Fixed an unlikely race condition of freeing measurements.

#### API changes

- The old *FLIM* class is replaced by a new implementation: *Flim*. In case you need the old implementation, there is a 1 to 1 replacement, see [here](#).
- All methods and measurements now throw exceptions instead of warning on wrong arguments like invalid channels or out-of-range parameters.
- Automatically call *freeTimeTagger* on *del/clear/Dispose* in Python/Matlab/LabVIEW/C#.
- Removed the *freeAllTimeTagger* method.
- Deprecate the multiple use of *createTimeTagger()* for one physical device. Pass on the *timetagger* object instead.
- *\_Log* is renamed to *LogBase*.
- Our libraries are compiled with VS 2019 now, so at least version 142 of the VC runtime is required in the final application.

## 11.7 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 *SynchronizedMeasurements* and *.getTagger()*.
- The last datapoint from a scope measurement is not marked as invalid any more.

## 11.8 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 available via TTCoincidenceTimestamp.

### Linux

- Fix for slow Linux device opening.

## 11.9 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.

## 11.10 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 Coincidencees 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

## 11.11 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 configurable 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

## 11.12 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
- Clarify 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

## 11.13 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/ $\mu$ s/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  $\leq 20$  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

## 11.14 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

## 11.15 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

## 11.16 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  $\geq 1.2$

### 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

## 11.17 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

## 11.18 V2.2.2 - 13.11.2018

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

## 11.19 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 ‘Conditional 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

## 11.20 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

## 11.21 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

## 11.22 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

## 11.23 V2.1.0 - 06.03.2018

#### Time Tagger Ultra

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

## 11.24 V2.0.4 - 01.02.2018

#### Bug fixes

- Closing the web application server window works properly now

## 11.25 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.

## 11.26 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

## 11.27 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

## 11.28 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

## 11.29 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

## 11.30 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

## 11.31 V1.0.0

initial release supporting python

## 11.32 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.



## A

All (*ChannelEdge* attribute), 34  
 autoCalibration() (*TimeTagger* method), 42  
 Average (*CoincidenceTimestamp* attribute), 34

## B

bins (*FlimFrameInfo* attribute), 71  
 built-in function  
     createTimeTagger(), 35  
     createTimeTaggerVirtual(), 35  
     freeTimeTagger(), 35  
     getTimeTaggerChannelNumberScheme(), 36  
     scanTimeTagger(), 35  
     setLogger(), 35  
     setTimeTaggerChannelNumberScheme(), 36

## C

CHANNEL\_UNUSED (*built-in variable*), 33  
 ChannelEdge (*built-in class*), 34  
 clear(), 54  
 clear() (*Dump* method), 78  
 clear() (*SynchronizedMeasurements* method), 80  
 clearConditionalFilter() (*TimeTagger* method), 38  
 clearOverflows() (*TimeTagger* method), 40  
 Coincidence (*built-in class*), 46  
 Coincidences (*built-in class*), 47  
 CoincidenceTimestamp (*built-in class*), 34  
 Combiner (*built-in class*), 46  
 ConstantFractionDiscriminator (*built-in class*), 50  
 Correlation (*built-in class*), 63  
 CountBetweenMarkers (*built-in class*), 57  
 Counter (*built-in class*), 56  
 Countrate (*built-in class*), 55  
 createTimeTagger()  
     built-in function, 35  
 createTimeTaggerVirtual()  
     built-in function, 35  
 CustomMeasurement (*built-in class*), 81

## D

DelayedChannel (*built-in class*), 49  
 Dump (*built-in class*), 78

## E

Error (*OverflowType* attribute), 74  
 Event (*built-in class*), 79  
 EventGenerator (*built-in class*), 51  
 External delay, 86

## F

Falling (*ChannelEdge* attribute), 34  
 FileReader (*built-in class*), 77  
 FileWriter (*built-in class*), 76  
 First (*CoincidenceTimestamp* attribute), 34  
 Flim (*built-in class*), 68  
 FlimBase (*built-in class*), 72  
 FlimFrameInfo (*built-in class*), 71  
 frame\_number (*FlimFrameInfo* attribute), 71  
 frameReady() (*Flim* method), 71  
 frameReady() (*FlimBase* method), 73  
 freeTimeTagger()  
     built-in function, 35  
 FrequencyMultiplier (*built-in class*), 48

## G

GatedChannel (*built-in class*), 48  
 getBinEdges() (*HistogramLogBins* method), 61  
 getBinWidths() (*CountBetweenMarkers* method), 57  
 getCaptureDuration(), 54  
 getChannel() (*VirtualChannel* method), 45  
 getChannelAbove() (*TriggerOnCountrate* method), 52  
 getChannelBelow() (*TriggerOnCountrate* method), 52  
 getChannelList() (*TimeTagger* method), 41  
 getChannels() (*TimeTagStreamBuffer* method), 75  
 getChannels() (*TriggerOnCountrate* method), 52  
 getChannels() (*VirtualChannel* method), 45  
 getConditionalFilterFiltered() (*TimeTagger* method), 38

`getConditionalFilterTrigger()` (*TimeTagger method*), 38

`getConfiguration()` (*FileReader method*), 77

`getConfiguration()` (*TimeTagger method*), 43

`getCounts()` (*TimeDifferences method*), 65

`getCountsTotal()` (*Countrate method*), 55

`getCurrentCountrate()` (*TriggerOnCountrate method*), 52

`getCurrentFrame()` (*Flim method*), 69

`getCurrentFrameEx()` (*Flim method*), 69

`getCurrentFrameIntensity()` (*Flim method*), 69

`getDACRange()` (*TimeTagger method*), 41

`getData()` (*Correlation method*), 63

`getData()` (*CountBetweenMarkers method*), 57

`getData()` (*Counter method*), 56

`getData()` (*Countrate method*), 55

`getData()` (*FileReader method*), 77

`getData()` (*Histogram method*), 59

`getData()` (*Histogram2D method*), 62

`getData()` (*HistogramLogBins method*), 61

`getData()` (*Scope method*), 79

`getData()` (*StartStop method*), 58

`getData()` (*TimeDifferences method*), 65

`getData()` (*TimeTagStream method*), 74

`getDataNormalized()` (*Correlation method*), 63

`getDataNormalizedCountsPerPs()` (*HistogramLogBins method*), 61

`getDataNormalizedG2()` (*HistogramLogBins method*), 61

`getDeadtime()` (*TimeTagger method*), 39

`getDelayHardware()` (*TimeTagger method*), 37

`getDelaySoftware()` (*TimeTagger method*), 37

`getDistributionCount()` (*TimeTagger method*), 42

`getDistributionPSec()` (*TimeTagger method*), 42

`getEventDivider()` (*TimeTagger method*), 39

`getEventTypes()` (*TimeTagStreamBuffer method*), 75

`getFence()` (*TimeTagger method*), 40

`getFrameNumber()` (*FlimFrameInfo method*), 71

`getFramesAcquired()` (*Flim method*), 69

`getHardwareDelayCompensation()` (*TimeTagger method*), 38

`getHistograms()` (*FlimFrameInfo method*), 72

`getIndex()` (*Correlation method*), 63

`getIndex()` (*CountBetweenMarkers method*), 57

`getIndex()` (*Counter method*), 56

`getIndex()` (*Flim method*), 69

`getIndex()` (*Histogram method*), 59

`getIndex()` (*Histogram2D method*), 62

`getIndex()` (*TimeDifferences method*), 65

`getIndex_1()` (*Histogram2D method*), 62

`getIndex_2()` (*Histogram2D method*), 62

`getInputDelay()` (*TimeTagger method*), 37

`getIntensities()` (*FlimFrameInfo method*), 72

`getInvertedChannel()` (*TimeTagger method*), 41

`getMaxFileSize()` (*FileWriter method*), 76

`getMissedEvents()` (*TimeTagStreamBuffer method*), 75

`getModel()` (*TimeTagger method*), 40

`getNormalization()` (*TimeTagger method*), 39

`getOverflows()` (*TimeTagger method*), 40

`getOverflows()` (*TimeTagStreamBuffer method*), 75

`getOverflowsAndClear()` (*TimeTagger method*), 40

`getPcbVersion()` (*TimeTagger method*), 41

`getPixelBegins()` (*FlimFrameInfo method*), 72

`getPixelEnds()` (*FlimFrameInfo method*), 72

`getPixelPosition()` (*FlimFrameInfo method*), 72

`getPsPerClock()` (*TimeTagger method*), 42

`getReadyFrame()` (*Flim method*), 69

`getReadyFrameEx()` (*Flim method*), 69

`getReadyFrameIntensity()` (*Flim method*), 70

`getReplaySpeed()` (*TimeTaggerVirtual method*), 45

`getSensorData()` (*TimeTagger method*), 43

`getSerial()` (*TimeTagger method*), 40

`getSummedCounts()` (*FlimFrameInfo method*), 72

`getSummedFrames()` (*Flim method*), 70

`getSummedFramesEx()` (*Flim method*), 70

`getSummedFramesIntensity()` (*Flim method*), 70

`getTagger()` (*SynchronizedMeasurements method*), 80

`getTestSignal()` (*TimeTagger method*), 40

`getTestSignalDivider()` (*TimeTagger method*), 43

`getTimestamps()` (*TimeTagStreamBuffer method*), 75

`getTimeTaggerChannelNumberScheme()` built-in function, 36

`getTotalEvents()` (*FileWriter method*), 76

`getTotalSize()` (*FileWriter method*), 76

`getTriggerLevel()` (*TimeTagger method*), 36

## H

Hardware delay, 86

`hasData()` (*FileReader method*), 77

`hasOverflows()` (*TimeTagStreamBuffer method*), 75

HIGH (*State attribute*), 79

HighResA (*Resolution attribute*), 33

HighResAll (*ChannelEdge attribute*), 34

HighResB (*Resolution attribute*), 33

HighResC (*Resolution attribute*), 34

HighResFalling (*ChannelEdge attribute*), 34

HighResRising (*ChannelEdge attribute*), 34

Histogram (built-in class), 59

Histogram2D (built-in class), 61

HistogramLogBins (built-in class), 60



## I

injectCurrentState() (*TriggerOnCountrate method*), 52  
 Input time stamp, 86  
 isAbove() (*TriggerOnCountrate method*), 52  
 isAcquiring() (*Flim method*), 70  
 isAcquiring() (*FlimBase method*), 73  
 isBelow() (*TriggerOnCountrate method*), 52  
 isChannelUnused() (*TimeTagger method*), 42  
 isRunning(), 54  
 isRunning() (*SynchronizedMeasurements method*), 80  
 isValid() (*FlimFrameInfo method*), 71  
 IteratorBase (*built-in class*), 91

## L

Last (*CoincidenceTimestamp attribute*), 34  
 ListedFirst (*CoincidenceTimestamp attribute*), 34  
 LOW (*State attribute*), 79

## M

MissedEvents (*OverflowType attribute*), 74

## O

on\_frame\_end() (*Flim method*), 71  
 on\_frame\_end() (*FlimBase method*), 73  
 OverflowBegin (*OverflowType attribute*), 74  
 OverflowEnd (*OverflowType attribute*), 74  
 OverflowType (*built-in class*), 74

## P

pixel\_count (*FlimFrameInfo attribute*), 71  
 pixels (*FlimFrameInfo attribute*), 71  
 process() (*CustomMeasurement method*), 81

## R

ready() (*CountBetweenMarkers method*), 58  
 ready() (*TimeDifferences method*), 65  
 registerChannel() (*TimeTagger method*), 41  
 registerMeasurement() (*SynchronizedMeasurements method*), 79  
 replay() (*TimeTaggerVirtual method*), 44  
 reset() (*TimeTagger method*), 36  
 Resolution (*built-in class*), 33  
 Rising (*ChannelEdge attribute*), 34

## S

scanTimeTagger()  
     built-in function, 35  
 Scope (*built-in class*), 79  
 setConditionalFilter() (*TimeTagger method*), 38  
 setDeadtime() (*TimeTagger method*), 39

setDelay() (*DelayedChannel method*), 49  
 setDelayHardware() (*TimeTagger method*), 36  
 setDelaySoftware() (*TimeTagger method*), 37  
 setEventDivider() (*TimeTagger method*), 38  
 setHardwareBufferSize() (*TimeTagger method*), 42  
 setInputDelay() (*TimeTagger method*), 37  
 setLED() (*TimeTagger method*), 43  
 setLogger()  
     built-in function, 35  
 setMaxCounts() (*TimeDifferences method*), 65  
 setMaxFileSize() (*FileWriter method*), 76  
 setNormalization() (*TimeTagger method*), 39  
 setReplaySpeed() (*TimeTaggerVirtual method*), 44  
 setSoundFrequency() (*TimeTagger method*), 43  
 setStreamBlockSize() (*TimeTagger method*), 42  
 setTestSignal() (*TimeTagger method*), 39  
 setTestSignalDivider() (*TimeTagger method*), 42  
 setTimeTaggerChannelNumberScheme()  
     built-in function, 36  
 setTriggerLevel() (*TimeTagger method*), 36  
 split() (*FileWriter method*), 76  
 Standard (*Resolution attribute*), 33  
 StandardAll (*ChannelEdge attribute*), 34  
 StandardFalling (*ChannelEdge attribute*), 34  
 StandardRising (*ChannelEdge attribute*), 34  
 start(), 54  
 start() (*SynchronizedMeasurements method*), 80  
 startFor(), 54  
 startFor() (*SynchronizedMeasurements method*), 80  
 StartStop (*built-in class*), 58  
 State (*built-in class*), 79  
 state (*Event attribute*), 79  
 stop(), 54  
 stop() (*Dump method*), 78  
 stop() (*SynchronizedMeasurements method*), 80  
 stop() (*TimeTaggerVirtual method*), 44  
 sync() (*TimeTagger method*), 41  
 SynchronizedMeasurements (*built-in class*), 79

## T

TDC time stamp, 86  
 time (*Event attribute*), 79  
 TimeDifferences (*built-in class*), 64  
 TimeDifferencesND (*built-in class*), 66  
 TimeTag (*OverflowType attribute*), 74  
 TimeTagger (*built-in class*), 36  
 TimeTaggerVirtual (*built-in class*), 44  
 TimeTagStream (*built-in class*), 74  
 TimeTagStreamBuffer (*built-in class*), 74  
 TriggerOnCountrate (*built-in class*), 52

## U

UNKNOWN (*State attribute*), [79](#)

unregisterChannel() (*TimeTagger method*), [41](#)

unregisterMeasurement() (*SynchronizedMeasurements method*), [80](#)

## W

waitForCompletion() (*TimeTaggerVirtual method*), [44](#)

waitForFence() (*TimeTagger method*), [40](#)

waitUntilFinished(), [54](#)