Time Tagger User Manual

Release 1.2.3-local-build

Swabian Instruments

Feb 22, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED

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

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

Time Tagger software download

https://www.swabianinstruments.com/time-tagger/downloads/

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 `Init` (select resolution if available) 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* you are encouraged to consult the following resources.

1. Check out the *Application Programmer's Interface* 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 `.\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\1-Quickstart` 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:

1. Connect your Time Tagger
2. Start the built-in test signal (~0.8 MHz square wave) and apply it to channels 1 and 2
3. Control the trigger level of your inputs - although it is not necessary here
4. Initialize a standard measurement (*`Correlation`*) in order to find the delay of the test signal between channel 1 and 2
5. 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
Basic software control (folder *1-basic_software_control*)		
1-A	Create a measurement	`createTimeTagger()`, `Counter.getData()`, `Counter.getIndex()`
	Count rate trace	`Counter`
1-B	Start & stop measurements	`Countrate`, `start()`, `stop()`, `startFor()`
1-C	Synchronize measurements	`SynchronizedMeasurements`
	Use different histograms	`Correlation`, `Histogram`, `StartStop`, `HistogramLogBins`
1-D	Virtual Channels	`DelayedChannel`, `Coincidence`, `GatedChannel`
1-E	Logging errors	`setLogger()`
1-F	(External) software clock	`TimeTaggerBase.setSoftwareClock()`, `FrequencyStability`
Controlling the hardware (folder *2-controlling-the-hardware*)		
2-A	Get hardware information	`scanTimeTagger()`, `TimeTagger.getSerial()`,
		`TimeTagger.getModel()`, `TimeTagger.getSensorData()`,
		`TimeTaggerBase.getConfiguration()`
2-B	The input trigger level	`TimeTagger.setTriggerLevel()`, `TimeTagger.getDACRange()`
2-C	Filter tags on hardware	`TimeTagger.setConditionalFilter()`, `TimeTagger.setEventDivider()`
2-D	Control input delays	`TimeTaggerBase.setInputDelay()`, `Histogram2D`
2-E	Overflows	`TimeTaggerBase.getOverflows()`, `TimeTagger.setTestSignalDivider()`
2-F	HighRes mode	`createTimeTagger()`, `TimeDifferences`
Dump and re-analyze time-tags (folder *3-dump-and-reanalyze-time-tags*)		
3-A	Dump tags by FileWriter	`FileWriter`
3-B	The Time Tagger Virtual	`createTimeTaggerVirtual()`, `TimeTaggerVirtual`
Working with raw time-tags (folder *4-working-with-raw-time-tags*)		
4-A	The FileReader	`FileReader`, `TimeTagStreamBuffer`
4-B	Streaming raw time-tags	`TimeTagStream`
4-C	Custom Measurements	`CustomMeasurement`

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 `TimeTagger` 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 prevent 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: Running the Example Suite requires '.NET Core 3.1 Desktop Runtime (v3.1.10)'.

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\xYZ\TimeTagger.lib`
- the library `.\driver\xYZ\TimeTagger.dll` is linked with the shared v142 or newer Visual Studio runtime (`/MD`)

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

2.1 Requirements

2.1.1 Operating System

Windows Windows 7 or higher

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

2.1.2 Installation

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

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

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

2.1.3 Web Application

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

2.1.4 Programming Examples

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

2.1.5 Linux

We do provide install packages and instructions for linux distributions too.

CHAPTER

THREE

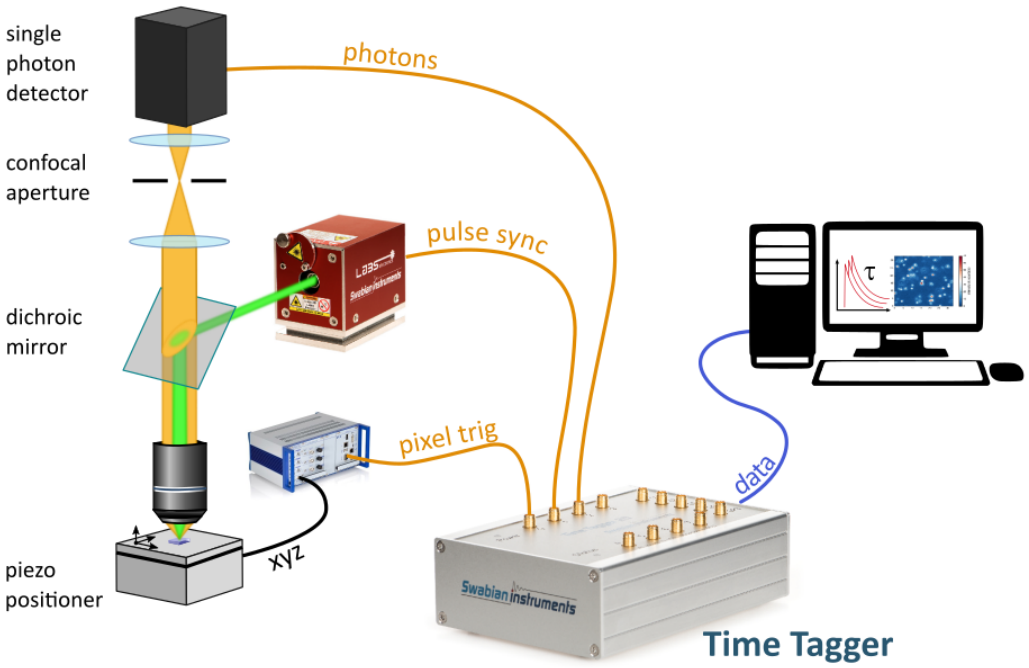
TUTORIALS

3.1 Confocal Fluorescence Microscope

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

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

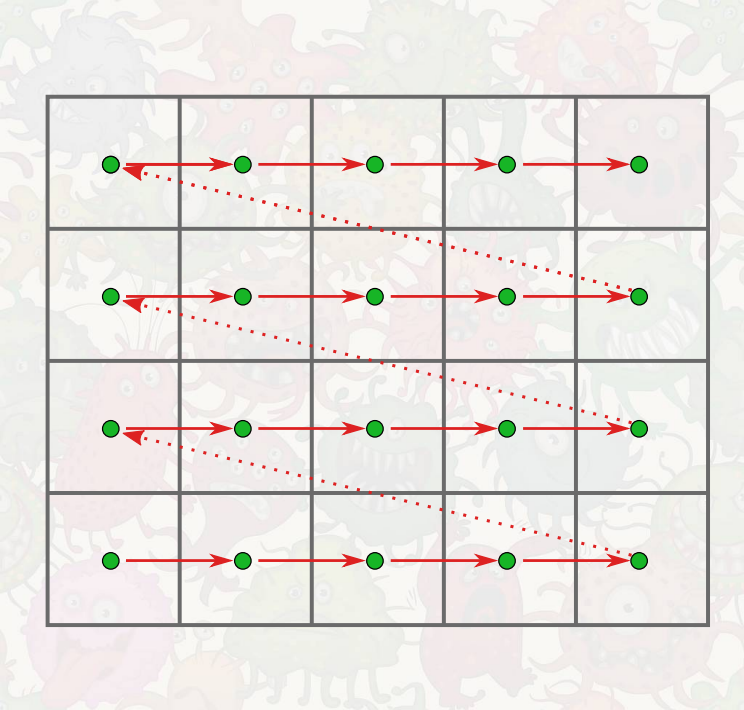
The following drawing shows a typical confocal fluorescence microscope setup.



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

Image from a raster scan

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



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

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

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

3.1.1 Time Tagger configuration

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

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

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

Now let's connect to the Time Tagger.

```
tt = createTimeTagger()
```

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

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

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

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

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

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

3.1.2 Intensity scanning microscope

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

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

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

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

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

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

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

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

script can continue immediately after starting the scan.

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

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

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

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

3.1.3 Fluorescence Lifetime Microscope

In the section *Intensity scanning microscope*, we completely discarded the time of arrival for photon and laser pulses. The Time Tagger allows you to record a fluorescence decay histogram for every pixel of the confocal image by taking into account the time difference between the arrival of the photon and laser pulses. This task can be achieved using the *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)

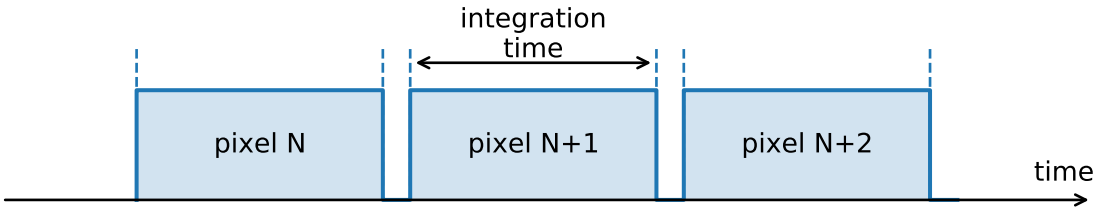
(continued from previous page)

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

3.1.4 Alternative pixel trigger formats

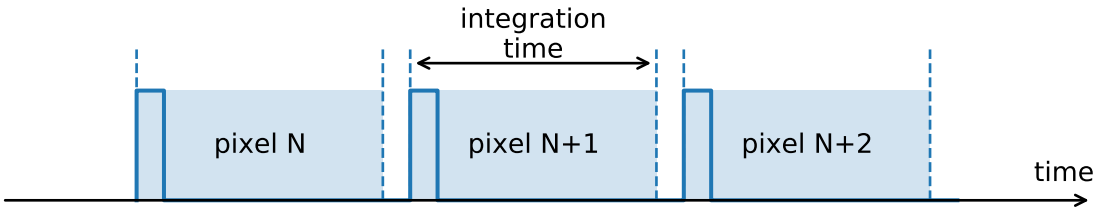
What if a piezo-scanner provides a different trigger signal compared to considered in the previous sections? In this section, we look into a few common types of trigger signals and how to adapt our data acquisition to make them work.

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.

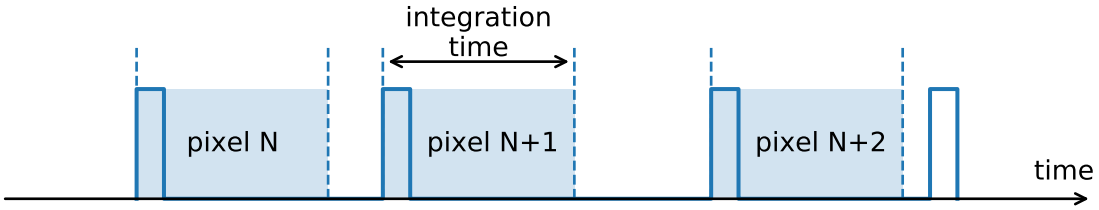
One can define the time window using the *DelayedChannel* which provides a delayed copy of the leading edge for the pixel pulse.

```
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

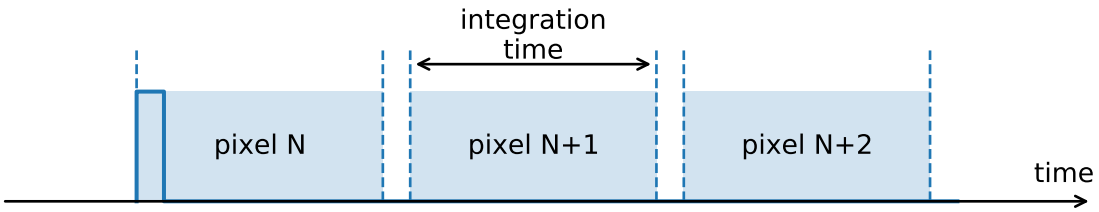


In some cases, a scanner generates the pixel pulses with no strictly defined period. However, most scanning measurements require constant integration time for every pixel. Compared to *CountBetweenMarkers*, the *TimeDifferences* measurement do not have a *PIXEL_END* marker and accumulate the histogram 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

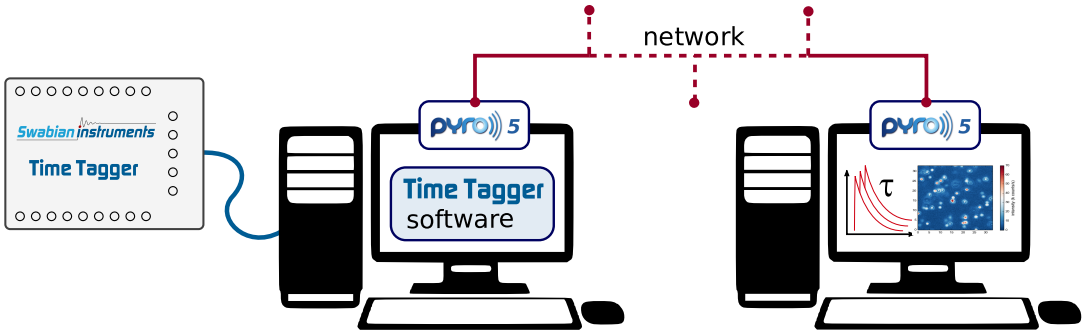


When a scanning system only has the line-start signal and does not provide the pixel pulses, we have to define time intervals for each pixel by other means. The pixel markers can be easily generated with *EventGenerator* virtual channel which generates events at times relative to the trigger event. Furthermore, 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, dtype='int64')
# 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.tolist())
pixel_stop_vch = EventGenerator(tt, LINE_START_CH, pixel_stop_times.tolist())

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.

3.2 Remote Time Tagger with Python



The Time Tagger is a great instrument for data acquisition whenever you detect, count, or analyze single photons. You can quickly set up a time correlation measurement, coincidence analysis, and much more. However, at some point in your project, you may want to control your experiment remotely. One option is to use remote desktop software like VNC, TeamViewer, Windows Remote Desktop, etc. What if you want to control your remote experiment programmatically? Are you using multiple computers and want to collect data from many of them at the same time? The solution for this is a remote control interface. Luckily, this task is very common and many software libraries cover the challenge of dealing with network sockets and messaging protocols.

Note: In the Time Tagger software version 2.10 we have introduced another way of controlling the Time Tagger device remotely, we call it *Network Time Tagger*. The Network Time Tagger implements server that sends the time-tag stream to the clients. Clients can connect to the server and run arbitrary measurements independently as if they are directly connected to the hardware device.

In contrast, this tutorial describes how to remotely control and create measurements that always run on the server only.

In this section, you will learn how to use Pyro5 and achieve seamless access to the *Time Tagger's API* remotely.

Listing 1: Teaser code

```
import matplotlib.pyplot as plt
from Pyro5.api import Proxy

TimeTagger = Proxy("PYRO:TimeTagger@server:23000")
tagger = TimeTagger.createTimeTagger()

hist = TimeTagger.Correlation(tagger, 1, 2, binwidth=5, n_bins=2000)
hist.startFor(int(10e12), clear=True)

x = hist.getIndex()
while hist.isRunning():
    plt.pause(0.1)
    y = hist.getData()
    plt.plot(x, y)
```

3.2.1 Remote procedure call

Remote procedure call (RPC) is a technology that allows interaction with remote programs by calling their procedures and receiving the responses. This involves a real code execution on one computer (server), while the client computer has only a substitute object (proxy) that mimics the real object running on the server. The proxy object knows how to send requests and data to the server and the server knows how to interpret these requests and how to execute the real code.

In the case of `Pyro5`, the proxy object and server code are provided by the library and we only need to tell `Pyro5` what we want to become available remotely.

3.2.2 Initial setup

You will need to have a Python 3.6 or newer installed on your computer. We recommend using Anaconda distribution.

Install the Time Tagger software if you have not done it yet. The description below assumes that you have the Time Tagger hardware and are familiar with the *Time Tagger API*.

The last missing part, the Pyro5 package, you can install from PyPi as

```
pip install Pyro5
```

3.2.3 Minimal example

Here we start from the simplest functional example and demonstrate working remote communication. The example consists of two parts: the server and the client code. You will need to run those in two separate command windows.

Server code

We need to create an adapter class with methods that we want to access remotely and decorate it with `Pyro5.api.expose()`. The following code is very simple. Later, we will extend it to expose more of the Time Tagger's functionality.

```
import Pyro5.api
import TimeTagger as TT

@Pyro5.api.expose
class TimeTaggerRPC:
    """Adapter for the Time Tagger Library"""

    def scanTimeTagger(self):
        """This method will become available remotely."""
        return TT.scanTimeTagger()


if __name__ == '__main__':
    # Start server and expose the TimeTaggerRPC class
    with Pyro5.api.Daemon(host='localhost', port=23000) as daemon:
        # Register class with Pyro
        uri = daemon.register(TimeTaggerRPC, 'TimeTagger')
        # Print the URI of the published object
        print(uri)
        # Start the server event loop
        daemon.requestLoop()
```

Client code

On the client side, we need to know the unique identifier of the exposed object, which was printed when you started the server. In Pyro5, every object is identified by a special string (URI) that contains the object identity string and the server address. As you can see in the code below, we do not use the Time Tagger software directly but rather communicate to the server that has it.

```
import Pyro5.api

# Connect to the TimeTaggerRPC object on the server
# This line is all we need to establish remote communication
TimeTagger = Pyro5.api.Proxy("PYRO:TimeTagger@localhost:23000")

# Now, we can call methods that will be executed on the server.
# Lets check what Time Taggers are available at the server
timetaggers = TimeTagger.scanTimeTagger()
print(timetaggers)

>> ['1740000ABC', '1750000ABC']
```

Congratulations! Now you have a very simple but functional communication to your remote Time Tagger software.

3.2.4 Creating the Time Tagger

By now, our code can communicate over a network and can only report the serial numbers of the connected Time Taggers. In this section, we will expand the server code and make it more useful. The next most important feature of the server is to expose the `createTimeTagger()` method to tell the server to initialize the Time Tagger hardware.

You may be tempted to extend the *TimeTaggerRPC* class as follows:

```
@Pyro5.api.expose
class TimeTaggerRPC:
    """Adapter for the Time Tagger Library"""

    def scanTimeTagger(self):
        """Return the serial numbers of the available Time Taggers."""
        return TT.scanTimeTagger()

    def createTimeTagger(self):
        """Create the Time Tagger."""
        return TT.createTimeTagger()  # This will fail! :(
```

To our great disappointment, the *createTimeTagger* method will fail when you try to access it from the client. The reason is in how the RPC communication works. The data and the program code have a certain format in which it is stored in the computer's memory, and this memory cannot be easily or safely accessed from a remote computer. The RPC communication overcomes this problem using data serialization, i.e., converting the data into a generalized format suitable for sending over a network and understandable by a client system.

The *Pyro5*, more specifically the *serpent* serializer it employs by default, knows how to serialize the standard Python data types like a list of strings returned by `scanTimeTagger()`. However, it has no idea how to interpret the `TimeTagger` object returned by the `createTimeTagger()`. Moreover, instead of sending the `TimeTagger` object to the client, we want to send a proxy object which allows the client to talk to the `TimeTagger` object on the server.

For the `TimeTagger`, we define an adapter class. Then we modify the *TimeTaggerRPC.createTimeTagger* to create an instance of the adapter class, register it with Pyro, and return it. Pyro will automatically take care of creating a proxy object for the client.

```
@Pyro5.api.expose
class TimeTagger:
    """Adapter for the Time Tagger object"""

    def __init__(self, args, kwargs):
        self._obj = TT.createTimeTagger(*args, **kwargs)

    def setTestSignal(self, *args):
        return self._obj.setTestSignal(*args)

    def getSerial(self):
        return self._obj.getSerial()

    # ... Other methods of the TT.TimeTagger class are omitted here.


@Pyro5.api.expose
class TimeTaggerRPC:
    """Adapter for the Time Tagger Library"""

    def scanTimeTagger(self):
        """Return the serial numbers of the available Time Taggers."""
```

(continues on next page)

(continued from previous page)

```
        return TT.scanTimeTagger()

    def createTimeTagger(self, *args, **kwargs):
        """Create the Time Tagger."""
        tagger = TimeTagger(args, kwargs)
        self._pyroDaemon.register(tagger)
        return tagger
        # Pyro will automatically create and send a proxy object
        # to the client.

    def freeTimeTagger(self, tagger_proxy):
        """Free Time Tagger. """
        # Client only has a proxy object.
        objectId = tagger_proxy._pyroUri.object
        # Get adapter object from the server.
        tagger = self._pyroDaemon.objectsById.get(objectId)
        self._pyroDaemon.unregister(tagger)
        return TT.freeTimeTagger(tagger._obj)
```

3.2.5 Measurements and virtual channels

By now, we can list available Time Tagger devices and create TimeTagger objects. The remaining part is to implement access to the measurements and virtual channels. We will use the same approach as with the TimeTagger class and create adapter classes for them.

```
@Pyro5.api.expose
class Correlation:
    """Adapter class for Correlation measurement."""

    def __init__(self, tagger, args, kwargs):
        self._obj = TT.Correlation(tagger._obj, *args, **kwargs)

    def start(self):
        return self._obj.start()

    def startFor(self, capture_duration, clear):
        return self._obj.startFor(capture_duration, clear=clear)

    def stop(self):
        return self._obj.stop()

    def clear(self):
        return self._obj.clear()

    def isRunning(self):
        return self._obj.isRunning()

    def getIndex(self):
        return self._obj.getIndex().tolist()

    def getData(self):
        return self._obj.getData().tolist()


@Pyro5.api.expose
```

(continues on next page)

(continued from previous page)

```
class DelayedChannel():
    """Adapter class for DelayedChannel."""

    def __init__(self, tagger, args, kwargs):
        self._obj = TT.DelayedChannel(tagger._obj, *args, **kwargs)

    def getChannel(self):
        return self._obj.getChannel()


@Pyro5.api.expose
class TimeTaggerRPC:
    """Adapter class for the Time Tagger Library"""

    #  Earlier code omitted (...)

    def Correlation(self, tagger_proxy, *args, **kwargs):
        """Create Correlation measurement."""
        objectId = tagger_proxy._pyroUri.object
        tagger = self._pyroDaemon.objectsById.get(objectId)
        pyro_obj = Correlation(tagger, args, kwargs)
        self._pyroDaemon.register(pyro_obj)
        return pyro_obj

    def DelayedChannel(self, tagger_proxy, *args, **kwargs):
        """Create DelayedChannel."""
        objectId = tagger_proxy._pyroUri.object
        tagger = self._pyroDaemon.objectsById.get(objectId)
        pyro_obj = DelayedChannel(tagger, args, kwargs)
        self._pyroDaemon.register(pyro_obj)
        return pyro_obj
```

Note: The methods *Correlation.getIndex()* and *Correlation.getData()* return `numpy.ndarray` arrays. Pyro5 does not know how to serialize `numpy.ndarray`, therefore for simplicity of the example, we convert them to the Python lists.

More efficient approach would be to register custom serializer functions for `numpy.ndarray` on both, server and client sides, see Customizing serialization section of the Pyro5 documentation.

3.2.6 Working example

Download the complete source files

- `simple_server.py`
- `simple_example.py`

Start the server in a terminal window:

```
> python simple_server.py
```

Now open a second terminal window and run the example:

```
> python simple_example.py
```

Let us take a look at the source code of the example (shown below). You may recognize that it is practically the same as using the Time Tagger package directly. The only difference is that the import statement `import TimeTagger` is replaced by the proxy object creation `TimeTagger = Pyro5.api.Proxy("PYRO:TimeTagger@localhost:23000")`.

Listing 2: simple_example.py

```
import numpy as np
import matplotlib.pyplot as plt
import Pyro5.api

TimeTagger = Pyro5.api.Proxy("PYRO:TimeTagger@localhost:23000")

# Create Time Tagger
tagger = TimeTagger.createTimeTagger()
tagger.setTestSignal(1, True)
tagger.setTestSignal(2, True)

print('Time Tagger serial:', tagger.getSerial())

hist = TimeTagger.Correlation(tagger, 1, 2, binwidth=2, n_bins=2000)
hist.startFor(int(10e12), clear=True)

fig, ax = plt.subplots()
# The time vector is fixed. No need to read it on every iteration.
x = np.array(hist.getIndex())
line, = ax.plot(x, x * 0)
ax.set_xlabel('Time (ps)')
ax.set_ylabel('Counts')
ax.set_title('Correlation histogram via Pyro-RPC')
while hist.isRunning():
    y = hist.getData()
    line.set_ydata(y)
    ax.set_ylim(np.min(y), np.max(y))
    plt.pause(0.1)

# Cleanup
TimeTagger.freeTimeTagger(tagger)
del hist
del tagger
del TimeTagger
```

See also:

The Time Tagger software installer includes more complete examples of the RPC server that includes more measurements, virtual channels and implements custom serialization of `numpy.ndarray` types. You can usually find the example files in the `C:\Program Files\Swabian Instruments\Time Tagger\examples\python\7-Remote-TimeTagger-with-Pyro5`.

3.2.7 What is next?

One can follow the ideas presented in this tutorial and implement a fully featured Python package. You can find an experimental version of such package at PyPi. Instead of manually wrapping every class and function of the Time Tagger API, the package employs metaprogramming and automatically generates adapter classes.

Let us know if you have any questions about RPC interface for the Time Tagger.

You can expand on the ideas presented in this tutorial, and implement remote control for your complete experiment.

CHAPTER

FOUR

SYNCHRONIZER

4.1 Overview

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

4.2 Requirements

Successful synchronization of your Time Taggers requires:

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

4.3 Cable connections

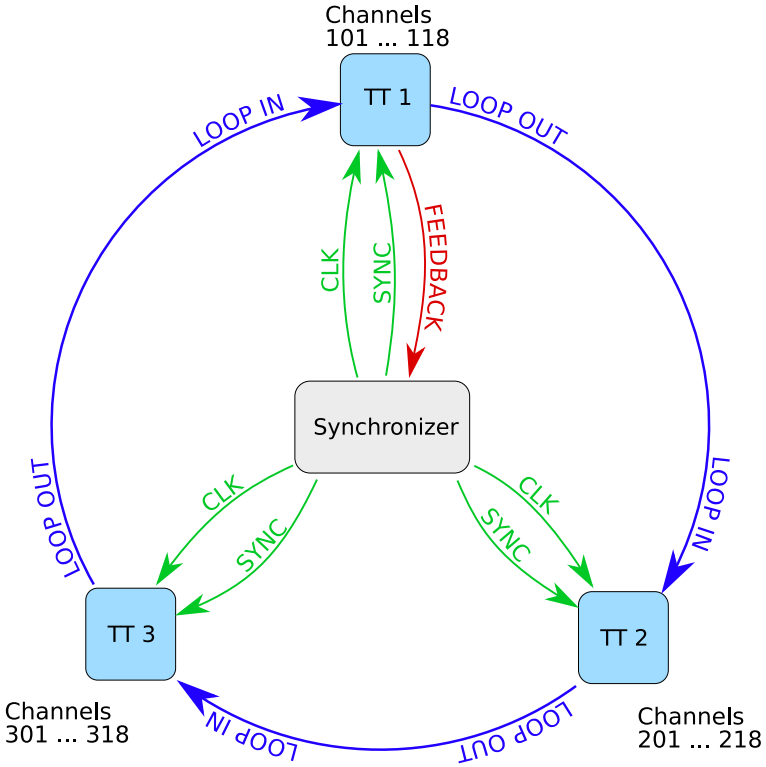
The Synchronizer provides a common clock signal for every Time Tagger as well as the synchronization signals. Furthermore, Time Taggers have to be connected to each other in a loop. The connection sequence in the loop defines the channel numbering order. An additional feedback signal is required to identify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

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

Table 1: Connections between the Synchronizer and Time Taggers

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

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



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

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

4.3.1 Using an external reference clock

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

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

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

Table 3: Signal parameters at *REF OUT*.

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

4.4 Software and channel numbering

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

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

```
tagger = createTimeTagger()
```

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

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

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

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

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

```
from TimeTagger import createTimeTagger, TT_CHANNEL_RISING_EDGES

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

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

4.4.1 Incomplete cable connections

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

4.4.2 Buffer overflows

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

4.5 Limitations

4.5.1 Conditional filter

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

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

4.5.2 Internal test signal

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

4.6 Status LEDs and troubleshooting

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

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

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

4.7 Synchronizer with only one Time Tagger

You can use the Synchronizer also with only one Time Tagger for two application scenarios:

4.7.1 Long term clock stability

The Synchronizer has a very good built-in clock which you can benefit from even when you have only one Time Tagger. Just connect any clock output of the Synchronizer to the clock input of the Time Tagger to benefit from the Synchronizer clock, which matters especially measuring long time differences.

4.7.2 Absolute clock timestamps

By connecting all signals from the Synchronizer as shown in *Cable connections* (LOOP IN and LOOP OUT must be shorted), the timestamps in the Time Tag stream will be referenced to the power-in time of the Synchronizer. Even when you disconnect from the Time Tagger, e.g., power down, USB timeout, the returned time tags are still referenced to the start time of the Synchronizer. To verify that this configuration is active, you will see a warning message in the console on `createTimeTagger()` that you are using the Synchronizer with only one Time Tagger.

CHAPTER

FIVE

HARDWARE

5.1 Input channels

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

5.1.1 Electrical characteristics

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

5.1.2 High Resolution Mode

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 converters (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.

5.2 Data connection

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

The data connection of the *Time Tagger Ultra* is USB 3.0. This allows to stream up to 70 M tags per second to the PC. 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 devices have LEDs showing status information. The "Power" LED turns green when the power is supplied to the device. An RGB 'Status' LED shows the information tabulated below.

Table 1: Status LED

Color	Description
green	firmware loaded
blinking green-orange	time tags are streaming
red flash (0.1 s)	an overflow occurred
continuous red	repeated overflows
blue	device initialization failed (check USB connection)

Table 2: 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)
blue	device not initialized, ext. clock disabled

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 `setInputDelay()`, do not apply to the Conditional Filter logic.

More details and explanations 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 - Time Tagger Ultra only

Note: An alternative and more flexible way to apply an external clock signal is the use of *`TimeTaggerBase.setSoftwareClock()`*. Since software version 2.10, the software clock is the recommended way of applying an external clock.

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.

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. This restriction does not exist for *`TimeTaggerBase.setSoftwareClock()`*.

5.12 Synchronization signals - Time Tagger Ultra only

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

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

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

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

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

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

CHAPTER

SIX

SOFTWARE OVERVIEW

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

6.1 Web application

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

6.2 Precompiled libraries and high-level language bindings

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

6.3 C++ API

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

CHAPTER

SEVEN

APPLICATION PROGRAMMER'S INTERFACE

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`.

7.1.4 Using Time Tagger remotely

Using Network Time Tagger you can stream the time-tags to a remote computer(s) and process them independently. You can easily work with your Time Tagger device over the network as if your remote computer is connected directly to the hardware. This example shows how you can start the server, connect a client to it and perform a simple countrate measurement.

You can start the server by calling *TimeTagger.startServer()* on a existing *TimeTagger* object.

```
# Connected to the hardware as usual
tagger = createTimeTagger()

# Start the server with full remote control enabled
tagger.startServer(AccessMode.Control)

# Keep this process running
input('Press ENTER to exit the server process...')

# Stop the server if user pressed ENTER key
tagger.stopServer()

# Disconnect from the hardware
freeTimeTagger(tagger)
```

For simplicity of the example we assume that the server is running as a separate process on the same computer. Therefore, we run the client code on the same computer and use `localhost` as a server address. You can also adjust the server address and try the client code on another PC.

```
# Server address, we assume it runs on the same computer
address = 'localhost'

# Connect to the server
ttn = createTimeTaggerNetwork(address)

# Enable test signal on the remote hardware
ttn.setTestSignal(1, True)
ttn.setTestSignal(2, True)

# Create `Countrate` measurement and run it for a fixed duration
cr = Countrate(ttn, [1,2,3])
cr.startFor(1e12)
cr.waitUntilFinished()

# Print the resulting data
print(cr.getData())

# Close the connection to the server
freeTimeTagger(ttn)
```

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 roll over 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.

class UsageStatisticsStatus

Disabled = 0
Usage statistics collection and upload is disabled.

Collecting = 1
Enable usage statistics collection local but without automatic uploading. This option might be useful to collect usage statistics for debugging purpose.

CollectingAndUploading = 2
Enable usage statistics collection and automatic upload

class AccessMode
This Enum controls how the Time Tagger server delivers the data-blocks to the connected clients, and if the clients are allowed to change the hardware settings.

Control
Clients have control over all settings on the Time Tagger. The data-blocks are delivered asynchronously to every client.

Listen
Clients cannot change settings on the Time Tagger and only subscribe to the exposed channels. The data-blocks are delivered asynchronously to every client.

SynchronousControl
The same as *AccessMode.Control* but the data is delivered synchronously to every client.

> **Warning:** This mode is not recommended for general use. The server will attempt to deliver a data-block to every connected client before sending the next data-block. Therefore, the data transmission will always be limited by the slowest client. If any of the clients cannot handle the data rate fast enough compared to the data-rate produced by the Time Tagger hardware, all connected clients will be affected and the Time Tagger hardware buffer may overflow. This can happen due to the network speed limit or insufficient CPU speed on any of the connected clients.

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*

Raises **RuntimeError** – if no Time Tagger devices are available or if the serial number is not correct.

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 different binwidths etc.

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

Returns TimeTaggerVirtual object

Return type *TimeTaggerVirtual*

createTimeTaggerNetwork([*address='localhost:41101'*])

Creates a new *TimeTaggerNetwork* object. During creation, the object tries to open a connection to the specified Time Tagger server that has been created by *TimeTagger.startServer()*. This makes the remote time-tag stream locally available. If the connection fails, the method will throw an exception.

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

Parameters **address** (*str*) – IP address, hostname, or domain-name of the server, where the Time Tagger server is running. The port number is optional and can be specified if server listens on a port other than default 41101.

Returns TimeTaggerNetwork object that can be used, e.g., for measurements

Return type *TimeTaggerNetwork*

Raises

- **RuntimeError** – if the connection to the server cannot be made.
- **ValueError** – if the address string has an invalid format.

getTimeTaggerServerInfo([*address='localhost:41101'*])

Returns TimeTagger configuration, exposed channels, hardware channels and virtual channels as a JSON formatted string.

Parameters **address** (*str*) – IP address, hostname or domain-name of the server, where the Time Tagger server is running. The port number is optional and can be specified if server listens on a port other than default 41101.

Returns Information about server, available channels and exposed channels.

Return type str or dict

Raises

- **RuntimeError** – if the connection to the server cannot be made.
- **ValueError** – if the address string has an invalid format.

freeTimeTagger(*tagger*)

Releases all Time Tagger resources and terminates the active connection.

Parameters **tagger** (TimeTaggerBase) – TimeTaggerBase 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]

scanTimeTaggerServers()
Scans the network for available Time Tagger servers.

Note: The server discovery algorithm uses multicast UDP messages sent to the address *239.255.255.83:41102*. This method is expected to work well in most situations, however there is a possibility when it could fail. The servers may not be discoverable if the system firewall rejects multicast traffic or blocks access to the used UDP port 41102, also multicast traffic is not usually forwarded to other IP networks by the routers.

Returns A list of addresses of the Time Tagger servers that are available in the network.

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

Usage statistics data collection

See also the section *Usage Statistics Collection*.

setUsageStatisticsStatus(*status*)

Parameters **status** (UsageStatisticsStatus) – New status of the usage statistics data collection.

This function allows a user to override the system-wide default setting on collection and submission of the usage statistics data. This function operates within the scope of a current OS user. The system-wide default setting is given during the installation of the Time Tagger software. Please run the installer again to allow collection and uploading or to disable the usage statistics.

getUsageStatisticsStatus()

Returns Returns the current status of the usage statistics for the current user. The status is described by the *UsageStatisticsStatus*.

Return type *UsageStatisticsStatus*

getUsageStatisticsReport()

This function returns the current state of the usage statistics report as a JSON formatted string. If there is no report data available or it was submitted just now, the output is a message: *Info: No report data available yet.* If you had given your consent earlier and then revoked it, this function will still return earlier accumulated report data.

Returns Usage statistics data encoded as JSON string.

Return type str

7.3 TimeTagger classes

The Time Tagger classes represent the different time-tag sources for your measurements and analysis. These objects are created by factory functions in the *Time Tagger library*:

Time Tagger hardware The *TimeTagger* represents a hardware device and allows access to hardware settings. To connect to a hardware Time Tagger and to get a *TimeTagger* object, use *createTimeTagger()*.

Virtual Time Tagger The *TimeTaggerVirtual* allows replaying files created with the *FileWriter*. To create a *TimeTaggerVirtual* object, use *createTimeTaggerVirtual()*.

Network Time Tagger The *TimeTaggerNetwork* allows the (remote) access to a Time Tagger made available via *TimeTagger.startServer()*. The *TimeTaggerNetwork* object is created with *createTimeTaggerNetwork()* which also establishes connection to the server.

7.3.1 General Time Tagger features

There is a set of common functionality for all Time Tagger objects that is bundled in the *TimeTaggerBase* class. The specific classes below inherit from *TimeTaggerBase*. Every *measurement* and *virtual channel* requires a reference to a *TimeTaggerBase* object with which it will be associated.

class TimeTaggerBase

setInputDelay(*channel*, *delay*)
This is a convenience method. It calls *setDelaySoftware()* if you use a Time Tagger 20 or the delay is > 2 µs, otherwise *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 *getDelaySoftware()* and *getDelayHardware()*.

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

Returns Delay time in picoseconds

Return type int

setDelayHardware(*channel*, *delay*)

Note: Method is only available for the Time Tagger Ultra.

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 also affects the *Conditional Filter*.

Parameters

- **channel** (*int*) – Channel number
- **delay** (*int*) – Delay time in picoseconds, the maximum/minimum value allowed is ±2000000 (±2 µs)

getDelayHardware(*channel*)

Note: Method is only available for the Time Tagger Ultra.

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

setDeadtime(*channel*, *deadtime*)
Sets the dead time of a channel in picoseconds. The requested time will be rounded to the nearest multiple of the internal clock period, 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 setting of the hardware deadtime filter (2 ns), an event arriving between 2 ns and 2.1 ns after the last event of that channel 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

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.

setSoftwareClock(*input_channel: int*, *input_frequency: float*, *averaging_periods: float = 1000*, *wait_until_locked: bool = True*)
Define in software one of the input channels as the base clock for all channels. This feature sets up a software PLL and rescales all incoming time-tags according to the software clock defined. The PLL provides a new timebase with "ideal clock tags" separated by exactly the defined *clock_period*. For measurements, you can use both, rescaled and ideal clock tags.

While the PLL is not locked, the time base of the instrument is invalid. In this case, the time-tag stream changes to the overflow mode. This means that after every call to `setSoftwareClock()`, you will find overflows because the PLL starts from an unlocked state.

> **Caution:** It is often useful to apply this feature in combination with `TimeTagger.setEventDivider()` on the *input_channel*. The values of *input_frequency* and *averaging_periods* correspond to the transferred time-tags, not to the physical frequency. Changing the *divider* independently after setting up the software clock may lead to a failure of the locking process. Do not add *input_channel* to the list of *filtered* channels in `TimeTagger.setConditionalFilter()`.
>
> For the Time Tagger 20, a phase error of 200 ps needs to be considered when using the software clock.

Parameters

- **input_channel** (*int*) – The physical channel that is used as software clock input.
- **input_frequency** (*int*) – The frequency of the software clock after application of `TimeTagger.setEventDivider()` (e.g. a 10 MHz clock signal with *divider = 20* has *input_frequency = 500 000*). The value should not deviate from the real frequency by more than a few percent. Default: 10E6, for 10 MHz.
- **averaging_periods** (*float*) – The number of cycles to average over. The suppression of discretization noise is improved by a higher *averaging_periods*. If the value is too large, however, this will result in increased phase jitter due to the drift of the internal clock or the applied software clock signal. Default: 1000.
- **wait_until_locked** (*bool*) – Blocks the execution until the software clock is locked. Throws an exception on locking errors. All locking log messages are filtered while this call is executed. Default: True

disableSoftwareClock()
Disable the software clock.

getSoftwareClockState()
Provides an object representing the current software clock state. This includes the configuration parameters as well as dynamic values generated based on the incoming signal.

Returns An object that contains the current state of the software clock.

Return type *SoftwareClockState*

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

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

isUnusedChannel(*channel*)

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

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

Returns True/False

Return type bool

getConfiguration()

Returns a JSON formatted string (dictionary in Python) containing complete information on the Time Tagger settings. It also includes descriptions of measurements and virtual channels created on this Time Tagger instance.

Returns Time Tagger settings and currently existing measurements.

Return type str or dict

7.3.2 Time Tagger hardware

class TimeTagger
Base class: *TimeTaggerBase*

This class provides access to the hardware and exposes methods to control hardware settings, such as trigger levels or even filters. Behind the scenes, it opens the USB connection, initializes the device and receives and manages the time-tag-stream.

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

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

This compensation can be understood as an implicit part of *setDelayHardware()* and *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 *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

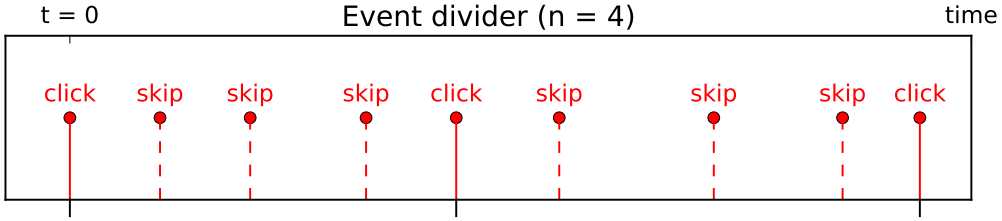
Return type list[int]

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. The divider is a 16 bit integer, so the maximum value is 65535.

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, max. 65535

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

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

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)

getChannelList(*type=ChannelEdge.All*)
Returns a list of channels corresponding to the given *type*.

Parameters **type** (ChannelEdge) – Limits the returned channels to the specified channel edge type

Returns List of channel numbers

Return type list[int]

setHardwareBufferSize(*size*)
Sets the maximum buffer size within the Time Tagger. 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.

Note: Time Tagger 20 uses by default the whole buffer of 8 MTags, which can be filled with a total data rate of up to 40 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 picoseconds 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.

startServer(*access_mode*, *channels=[]*, *port=41101*)
Start a Time Tagger server that can be accessed via *TimeTaggerNetwork*. The server access mode controls if the clients are allowed to change the hardware parameters. See also: *AccessMode*.

Parameters

- **access_mode** (AccessMode) – *AccessMode* in which the server should run. Either control or listen
- **channels** (*list[int]*) – Channels to be streamed. Used only when `access_mode=AccessMode.Listen`
- **port** (*int*) – Port at which this Time Tagger server will be listening on.

Raises **RuntimeError** – if server is already running.

stopServer()
Stop the Time Tagger server if currently running, otherwise do nothing.

isServerRunning()

Returns True is server is running and False otherwise.

Return type bool

7.3.3 The TimeTaggerVirtual class

In the Time Tagger software version 2.6.0, we have introduced the new *TimeTaggerVirtual*, which allows replaying 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

Base class: *TimeTaggerBase*

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

getConfiguration()
Returns a JSON formatted string (*dict* in Python) containing information on the TimeTaggerVirtual instance and on the real Time Tagger settings stored in the current time tag stream file.

7.3.4 The TimeTaggerNetwork class

In the Time Tagger software version 2.10, we have introduced a way of sending the time-tag stream to other applications and even remote computers for independent processing. We call this feature *Network Time Tagger*. You can use it with any Time Tagger hardware device by starting the time-tag stream server with *TimeTagger.startServer()*. Once the server is running, the clients can connect to it by calling *createTimeTaggerNetwork()* and specifying the server address. A client can be any computer that can access the server over the network or another process on the same computer. It is also possible to run the server and client on different operating systems or use different programming languages.

Note on performance

The Network Time Tagger server sends a time tag stream in a compressed format requiring about 4 bytes per time tag. Every client receives the data only from the channels required by the client. The maximum achievable data rate will depend on multiple factors, like server and client CPU performance, operating system, network adapter used, and network bandwidth, as well as the whole network infrastructure.

In a 1 Gbit Ethernet network, it is possible to achieve about 26 MTags/second of the total outgoing data rate from the server. Note that this bandwidth is shared among all clients connected. Likewise, a 10 Gbit Ethernet network allows reaching higher data rates while having more clients. In our tests, we reached up to 40 MTags/s per client.

When you run the server and the client on the same computer, the speed of the network adapters installed on your system becomes irrelevant. In this case, the operating system sends the data directly from the server to the client.

---

class TimeTaggerNetwork
Base class: *TimeTaggerBase*

Note: Although the *TimeTaggerNetwork* formally inherits from *TimeTaggerBase*, almost all methods of the hardware Time Tagger *TimeTagger* are available on the client (except for *TimeTagger.startServer()* and *TimeTagger.stopServer()*). These redundant methods are not listed in this section. A call to a method that exists on *TimeTagger* will be forwarded to the server. If a method with similar functionality exists on the *TimeTaggerNetwork* only, it can be distinguished by the suffix ... *Client*. If the server is running in *AccessMode.Listen* and a method call forwarded to the server would cause setting changes on the server-side, the call will raise an exception on the client.

This scheme of forwarding may lead to unexpected behavior: If the server is started in *AccessMode.Listen* with a restricted set of *channels* and you call *TimeTagger.getChannelList()* on the client side, not all channels returned by this method can be accessed. You can request the list of accessible channels from the server with *getTimeTaggerServerInfo()*.

The *TimeTaggerNetwork* represents a client-side of the Network Time Tagger and provides access to the Time Tagger server. A server can be created on any physical Time Tagger by calling *TimeTagger.startServer()*. The *TimeTaggerNetwork* object is created by calling *createTimeTaggerNetwork()*.

isConnected()
Check if the Network Time Tagger is currently connected to a server.

Returns True/False

Return_type bool

setDelayClient(*channel*, *delay*)
Sets an artificial software delay per channel on the client side. To specify it on the server side, see *setDelaySoftware()* or *setDelayHardware()* (Time Tagger Ultra only). This delay will be applied only on this object and will not affect the server settings or delays at any other clients connected to the same Time Tagger server.

Parameters

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

getDelayClient(*channel*)
Returns the value of the delay applied on the client-side in picoseconds for the specified channel.

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

Returns input delay in picoseconds

Return_type int

clearOverflowsClient()
Clears the overflow counter on the client-side. A call to *getOverflows()* will return the information as it is available on the server. See *getOverflowsClient()* for more information on client-side overflows.

getOverflowsClient()
If the server is not able to send all the time-tags to the client, e.g. due to limited network bandwidth, the time-tag stream switches to the overflow mode. This means that the client might experience additional overflow events that are not originating from the hardware. This counter counts all overflows occurred on the hardware and on the server since the client connection or last call to *clearOverflowsClient()* or *getOverflowsAndClearClient()*.

Returns The value of the client-side overflow counter.

Return_type int

getOverflowsAndClearClient()
The same as *getOverflowsClient()* but also clears the client-side counter. See *getOverflowsClient()* for more information on client-side overflows.

7.3.5 Additional classes

class SoftwareClockState
The *SoftwareClockState* object contains the current configuration state:

clock_period: int
The rounded clock period matching the input frequency set in *TimeTaggerBase.setSoftwareClock()*.

input_channel: int
The physical input channel of the software clock set in *TimeTaggerBase.setSoftwareClock()*.

ideal_clock_channel: `int`
A virtual channel number to receive the ideal clock tags. During a locking period, these tags are separated by *clock_period* by definition. To receive the rescaled measured clock tags, use *clock_channel*.

averaging_periods: `float`
The averaging periods set in `TimeTaggerBase.setSoftwareClock()`.

enabled: `bool`
Indicates whether the software clock is active or not.

Beyond the configuration state, the object provides current runtime information of the software clock:

is_locked: `bool`
Indicates whether the PLL of the software clock was able to lock to the input signal.

error_counter: `int`
Amount of locking errors since the last `TimeTaggerBase.setSoftwareClock()` call.

last_ideal_clock_event: `int`
Timestamp of the last ideal clock event in picoseconds.

period_error: `float`
Current deviation of the measured clock period from the ideal period given by *clock_period*.

phase_error_estimation: `float`
Current root of the squared differences of *clock_input* timestamps and ideal clock timestamps. This value includes the discretization noise of the *clock_input* channel.

7.4 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.

7.4.1 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.

7.4.2 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.

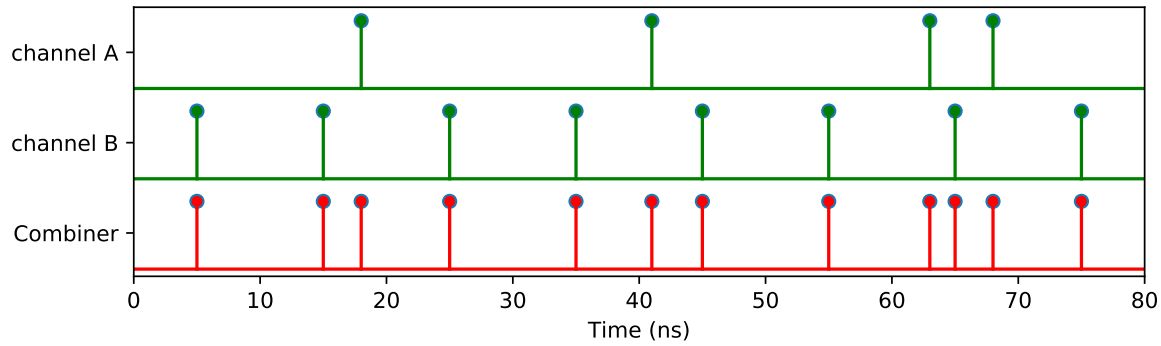
---

getConfiguration()
Returns configuration data of the virtual channel object. The configuration includes the name, values of the current parameters and the channel numbers. Information returned by this method is also provided with *TimeTaggerBase.getConfiguration()*.

Returns Configuration data of the virtual channel object.

Return type dict

7.4.3 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.

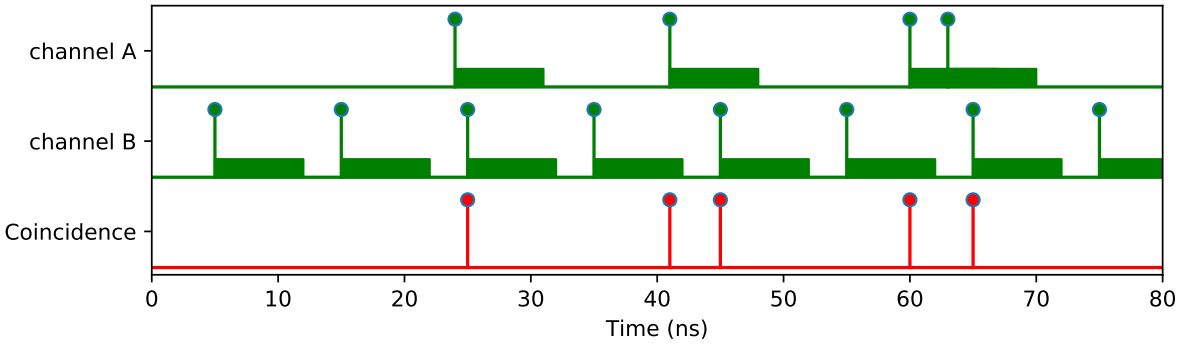
class Combiner(*tagger*, *channels=[]*)

Parameters

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

See all common methods

7.4.4 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** (TimeTaggerBase) – 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

See all common methods

7.4.5 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. The number of coincidence groups is limited to 64 per *Coincidences* object.

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 channel 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)
```

(continues on next page)

(continued from previous page)

```
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 channel 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).

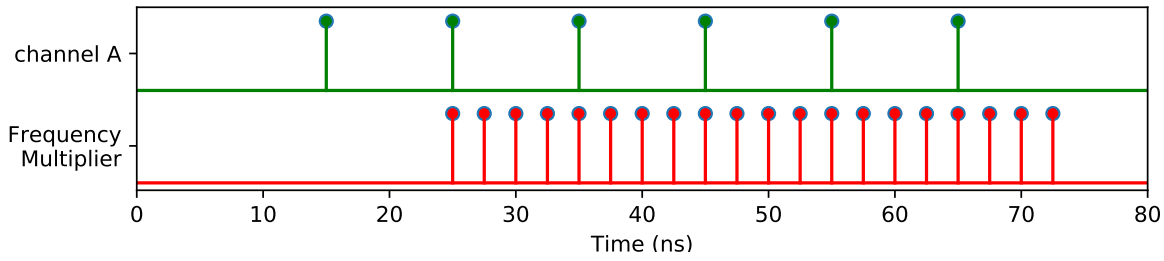
class Coincidences(*tagger*, *coincidenceGroups*, *coincidenceWindow*, *timestamp*)

Parameters

- **tagger** (TimeTaggerBase) – 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)

See all common methods

7.4.6 FrequencyMultiplier



Frequency Multiplier for a channel with a periodic signal.

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

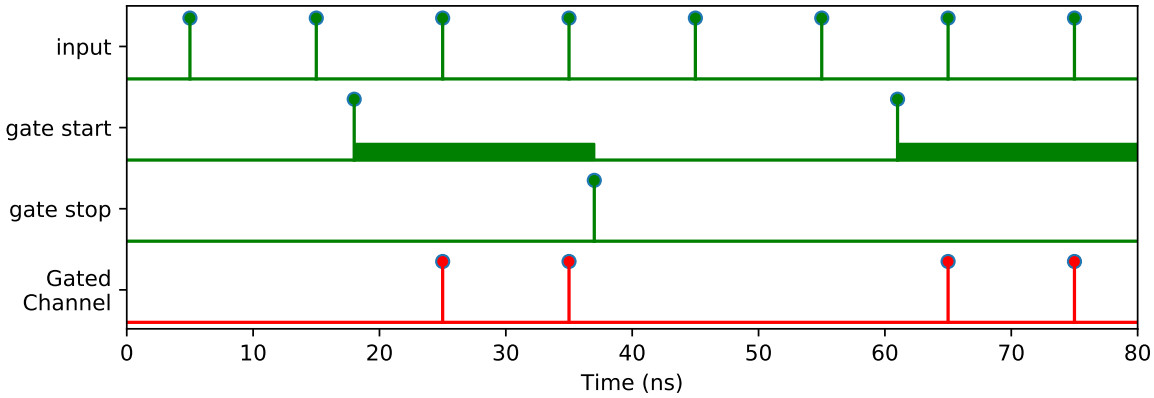
class FrequencyMultiplier(*tagger*, *input_channel*, *multiplier*)

Parameters

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

See all common methods

7.4.7 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*.

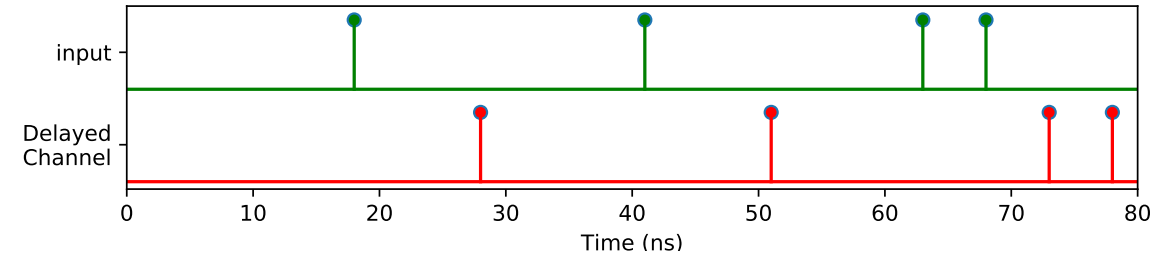
class GatedChannel(*tagger*, *input_channel*, *gate_start_channel*, *gate_stop_channel*)

Parameters

- **tagger** (TimeTaggerBase) – 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

See all common methods

7.4.8 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, *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** (*TimeTaggerBase*) – time tagger object
- **input_channel** (*int*) – channel to be delayed
- **delay** (*int*) – amount of time to delay in ps

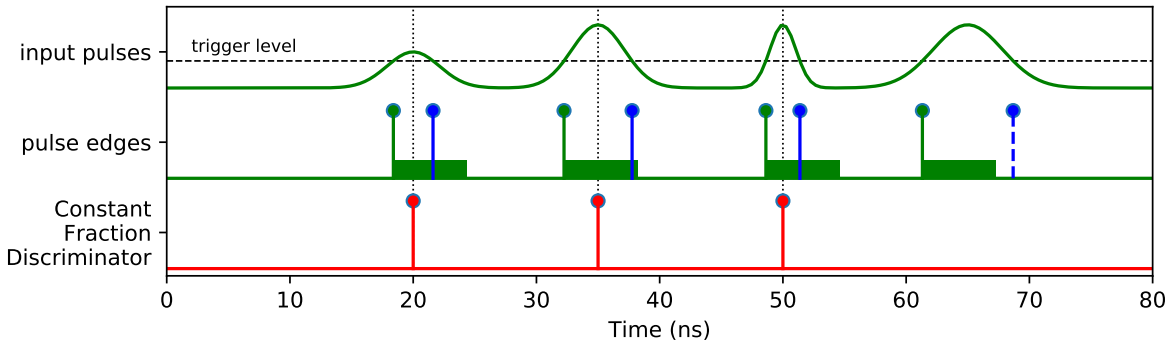
See all common methods

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

7.4.9 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. This leads to:

- The transferred data of the input channel is twice the regular input rate.
- When you shift the signal, e.g., via *setInputDelay()*, you have to shift both edges.
- When you use the conditional filter, apply the trigger from both channels.

---

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

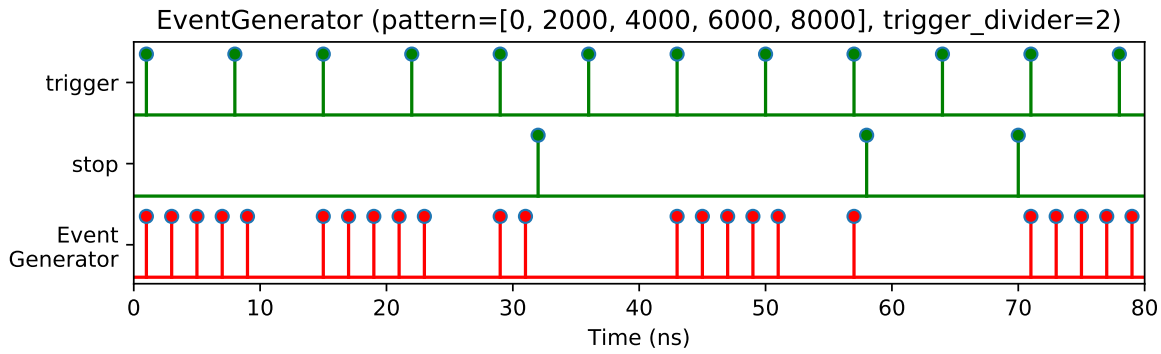
Parameters

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

- **channels** (*list[int]*) – list of channels on which to perform CFD
- **search_window** (*int*) – max pulse duration in picoseconds to be detected

See all common methods

7.4.10 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

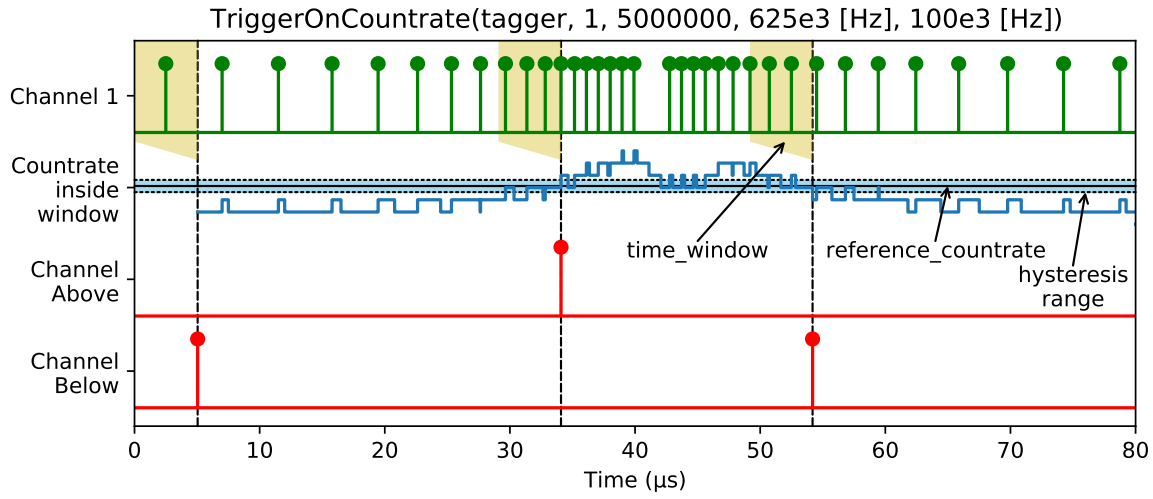
- **tagger** (*TimeTaggerBase*) – Time Tagger object instance.
- **trigger_channel** (*int*) – Channel number of the trigger signal.
- **pattern** (*list[int]*) – List of relative timestamps defining the pattern executed upon a trigger event.
- **trigger_divider** (*int*) – Factor by which the number 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)

See all common methods

7.4.11 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 *reference_countrate* + *hysteresis* or above. Vice versa, the *below* channel fires when the channel is in the *above* state and falls to the limit of *reference_countrate* - *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** (*TimeTaggerBase*) – 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* >= *reference_countrate* + *hysteresis*, whereas it is *countrate* <= *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.

See all common methods

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.

7.5 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 set 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.

7.5.1 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.

FrequencyStability Analyzes the frequency stability of period signals.

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

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

Histogram2D A 2-dimensional histogram of correlated time differences. This can be used in measurements similar to 2D NRM spectroscopy. (Single-Start, Single-Stop)

HistogramND A n-dimensional histogram of correlated time differences. (Single-Start, Single-Stop)

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. (Single-Start, Single-Stop)

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 on overview about the possibilities for the raw time-tag-stream access.

Sampler The *Sampler* class allows sampling the state of a set of channels via a trigger channel.

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

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

7.5.2 Common methods

class IteratorBase

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=- 1*)
Blocks the execution until the measurement has finished. Can be used with `startFor()`.

Parameters **timeout** (*int*) – timeout in milliseconds. Negative value 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

getConfiguration()

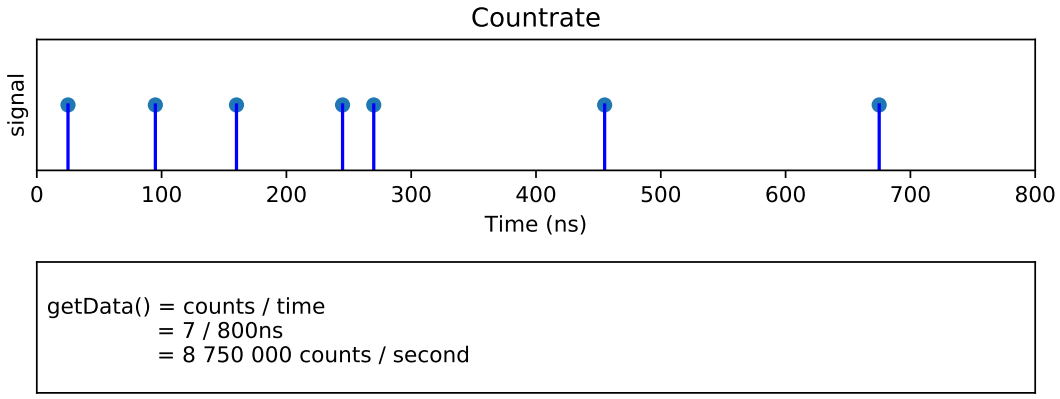
Returns configuration data of the measurement object. The configuration includes the measurement name, and the values of the current parameters. Information returned by this method is also provided with *TimeTaggerBase.getConfiguration()*.

Returns Configuration data of the measurement object.

Return type dict

7.5.3 Event counting

Countrate



Measures the average count rate on one or more channels. Specifically, it determines the counts per second on the specified channels starting from the very first tag arriving after the instantiation or last call to *clear()* of the measurement. The *Countrate* works correctly even when the USB transfer rate or backend processing capabilities are exceeded.

class Countrate(*tagger*, *channels*)

Parameters

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

See all common methods

getData()

Returns Average count rate in counts per second.

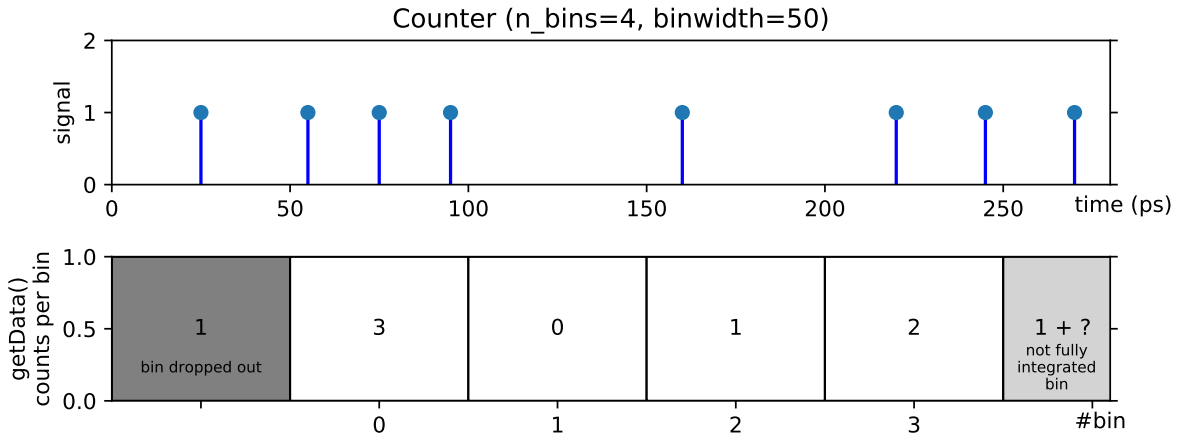
Return type 1D_array[float]

getCountsTotal()

Returns The total number of events since the instantiation of this object.

Return type 1D_array[int]

Counter



Time trace of the count rate on one or more channels. Specifically, this measurement repeatedly counts tags within a time interval *binwidth* and stores the results in a two-dimensional array of size *number of channels* by *n_values*. The incoming data is first accumulated in a not-accessible bin. When the integration time of this bin has passed, the accumulated data is added to the internal buffer, which can be accessed via the *getData...* methods. Data stored in the internal circular buffer is overwritten when *n_values* are exceeded. You can prevent this by automatically stopping the measurement in time as follows `counter.startFor(duration=binwidth*n_values)`.

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

Parameters

- **tagger** (`TimeTaggerBase`) – 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([*rolling=True*])

Returns an array of accumulated counter bins for each channel.

The optional parameter *rolling*, controls if the not integrated bins are padded before or after the integrated bins. This has observable effect only if the array is not yet completely filled. When `rolling=True`, the most recent data is stored in the last bin of the array and every new completed bin shifts all other bins right-to-left. When `rolling=False`, the most recent data is stored in the next bin after previous such that the array is filled up left-to-right. When array becomes full and the Counter is still running, the array will be shifted right-to-left and the latest count will be stored in the last bin.

Parameters **rolling** (*bool*) – Controls how the incomplete counter array is padded.

Returns An array of size 'number of channels' by *n_values* containing the counts in each fully integrated 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]

getDataNormalized([*rolling=True*])
Does the same as *getData()* but returns the count rate in Hz as a float. Not integrated bins and bins in overflow mode are marked as *NaN*.

Return type 2D_array[float]

getDataTotalCounts()
Returns total number of events per channel since the last call to *clear()*, including the currently integrating bin. This method works correctly even when the USB transfer rate or backend processing capabilities are exceeded.

Returns Number of events per channel.

Return type 1D_array[int]

getDataObject(*remove=False*)
Returns *CounterData* object containing a snapshot of the data accumulated in the *Counter* at the time this method is called.

Parameters **remove** (*bool*) – Controls if the returned data shall be removed from the internal buffer.

Returns An object providing access to a snapshot data.

Return type *CounterData*

class CounterData
Objects of this class are created and returned by *Counter.getDataObject()*, and contain a snapshot of the data accumulated by the *Counter* measurement.

size: int
Number of returned bins.

dropped_bins: int
Number of bins which have been dropped because *n_values* of the *Counter* has been exceeded.

overflow: bool
Status flag for whether any of the returned bins have been in overflow mode.

getIndex()

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

Return type 1D_array[int]

getData()

Returns An array of size 'number of channels' by *size* containing only fully integrated bins.

Return type 2D_array[int]

getDataNormalized()
Does the same as *getData()* but returns the count rate in counts/second. Bins in overflow mode are marked as *NaN*.

Return type 2D_array[float]

getDataTotalCounts()
Returns the total number of events per channel since the last call to *clear()*, excluding the counts of the internal bin where data is currently integrated into. This method works correctly even when the USB transfer rate or backend processing capabilities are exceeded.

Returns Number of events per channel.

Return type 1D_array[int]

getTime()

Returns A vector of size *size* containing the time corresponding to the return value of *CounterData.getData()* in ps.

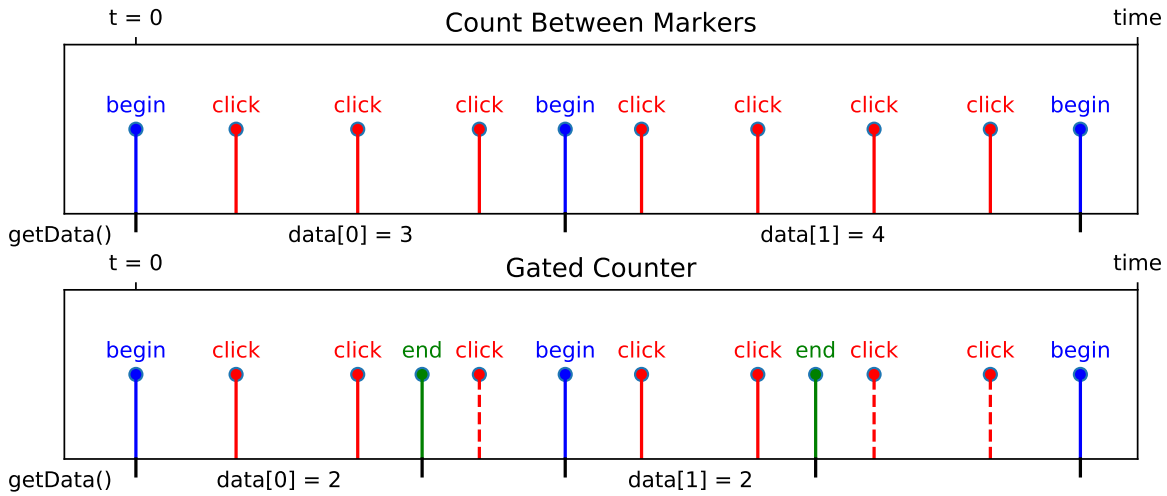
Return type 1D_array[int]

getOverflowMask()
Array of values for each bin that indicate if an overflow occurred during accumulation of the respective bin.

Returns An array of size *size* containing overflow mask.

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** (TimeTaggerBase) – 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()

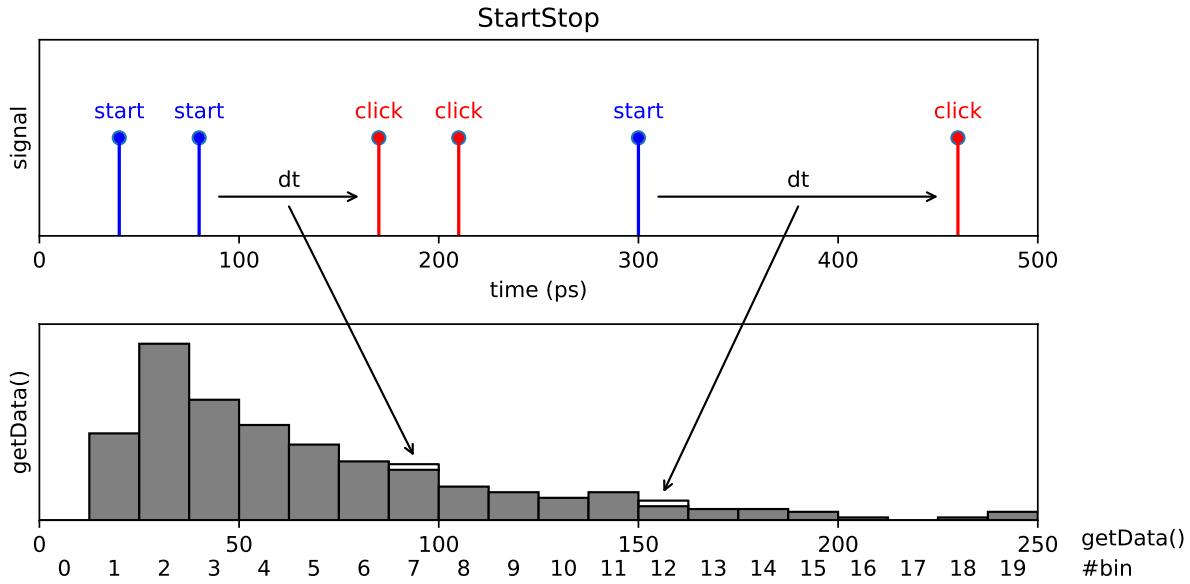
Returns True when the entire array is filled.

Return type bool

7.5.4 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** (TimeTaggerBase) – 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

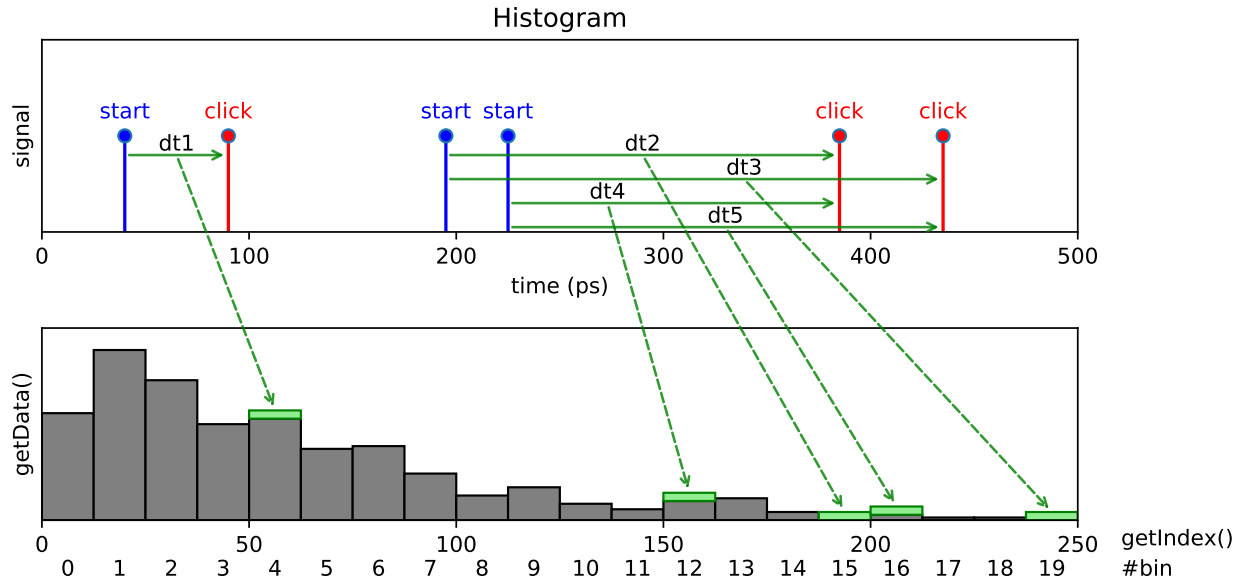
See all common methods

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** (TimeTaggerBase) – 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]

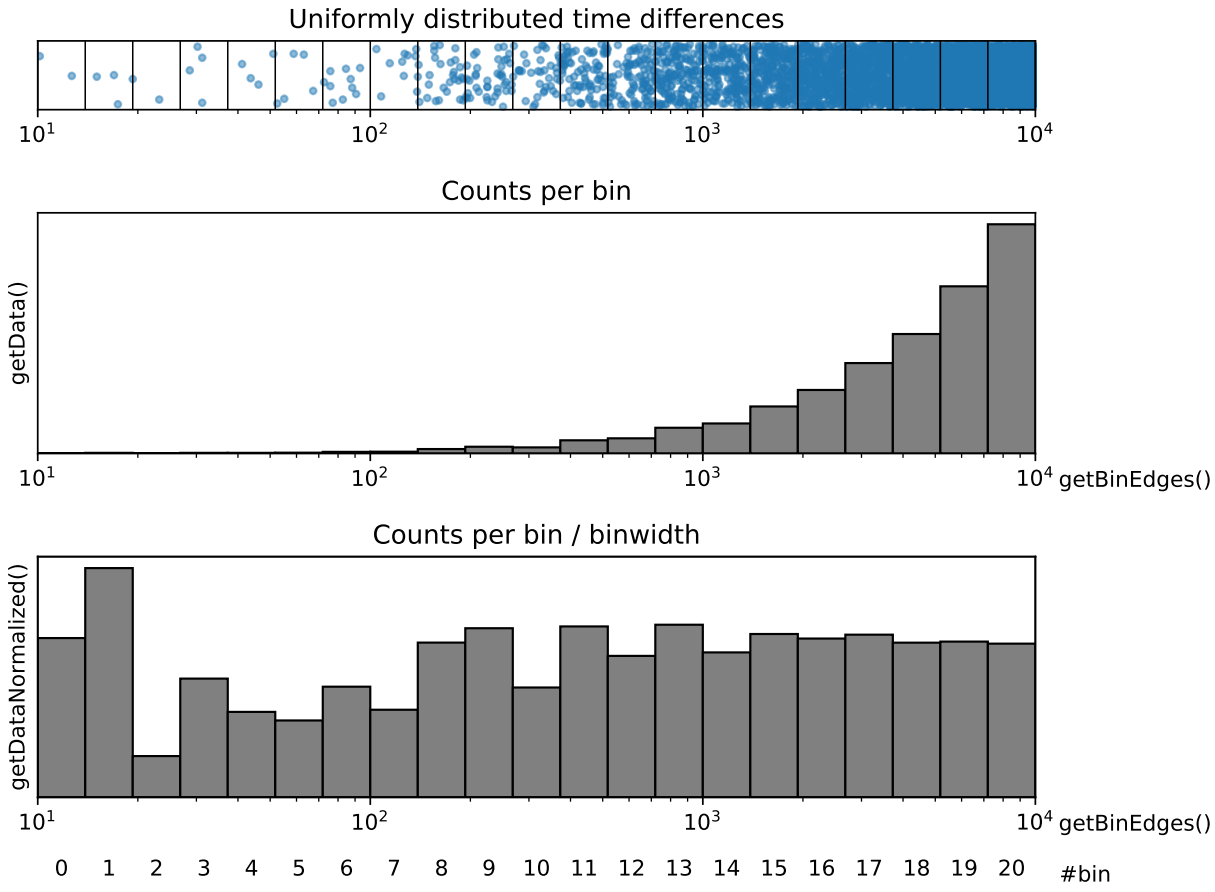
getIndex()

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** (TimeTaggerBase) – 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^exp_start` in seconds where the very first bin begins
- **exp_stop** (*float*) – exponent `10^exp_stop` in seconds where the very last bin ends
- **n_bins** (*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()*

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

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

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

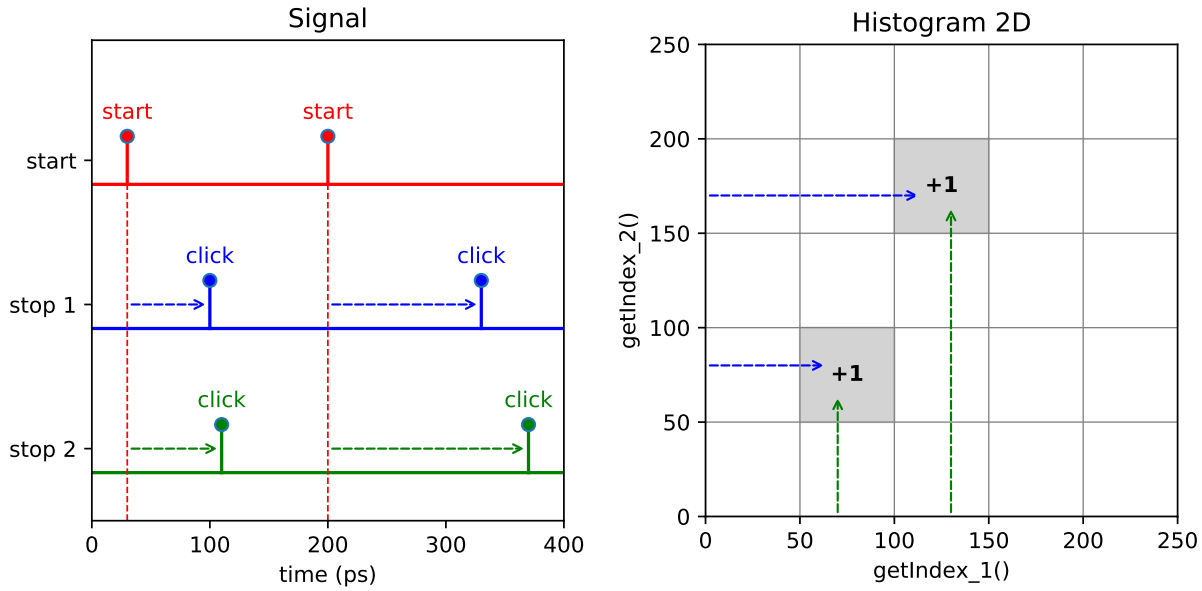
Return type 1D_array[float]

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 a 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. The first stop click of each axis is taken after the start click to evaluate the histogram counts.

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

Parameters

- **tagger** (TimeTaggerBase) – 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]

HistogramND

This measurement is the generalisation of *Histogram2D* to an arbitrary number of dimensions. The data within the histogram is acquired via a single-start, single-stop analysis for each axis. The first stop click of each axis is taken after the start click to evaluate the histogram counts.

HistogramND can be used as a 1D *Histogram* with single-start single-stop behavior.

class HistogramND(*tagger*, *start_channel*, *stop_channels*, *binwidths*, *n_bins*)

Parameters

- **tagger** (TimeTagger) – time tagger object
- **start_channel** (*int*) – channel on which start clicks are received

- **stop_channels** (*list[int]*) – channel list on which stop clicks are received defining the time axes
- **binwidths** (*list[int]*) – bin width in ps for the corresponding time axis
- **n_bins** (*list[int]*) – the number of bins along the corresponding time axis

See all common methods

getData()

Returns a one-dimensional array of the size of the product of n_bins containing the histogram data. The array order is in row-major. For example, with `stop_channels=[ch1, ch2]` and `n_bins=[2, 2]`, the 1D array would represent 2D bin indices in the order `[(0,0), (0,1), (1,0), (1,1)]`, with (index of `ch1`, index of `ch2`). Please reshape the 1D array to get the N-dimensional array. The following code demonstrates how to reshape the returned 1D array into multidimensional array using NumPy.

```
channels = [2, 3, 4, 5]
n_bins = [5, 3, 4, 6]
binwidths = [100, 100, 100, 50]
histogram_nd = HistogramND(tagger, 1, channels, binwidths, n_bins)
sleep(1)  # Wait to accumulate the data
data = histogram_nd.getData()
multidim_array = numpy.reshape(data, n_bins)
```

Returns Flattened array of histogram bins.

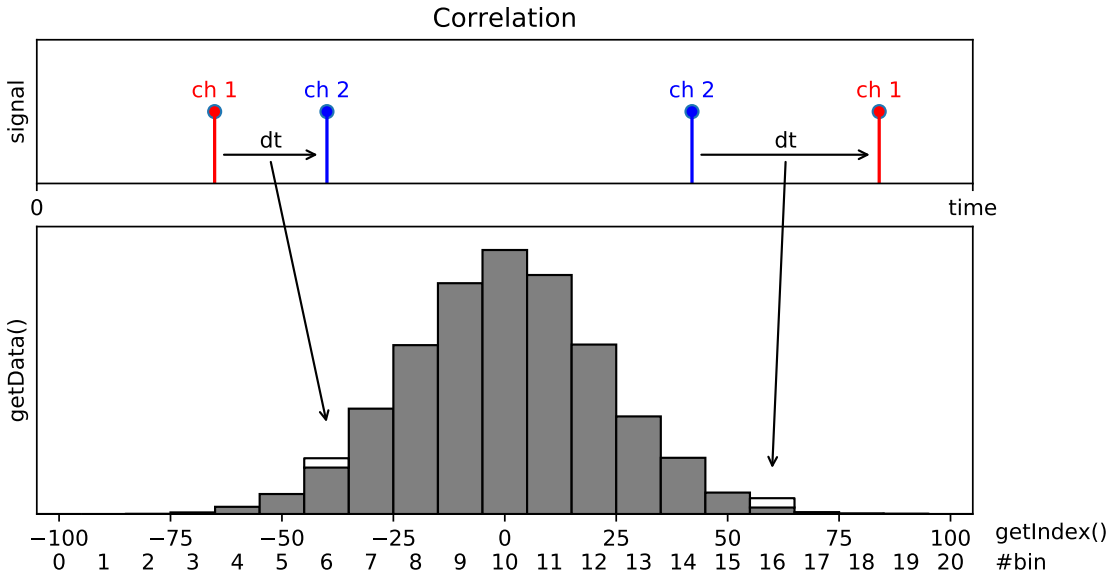
Return type 1D_array[int]

getIndex(*dim*)

Returns Returns a vector of size n_bins[dim] containing the bin locations in ps for the corresponding time axis.

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** (TimeTaggerBase) – 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:

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

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

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

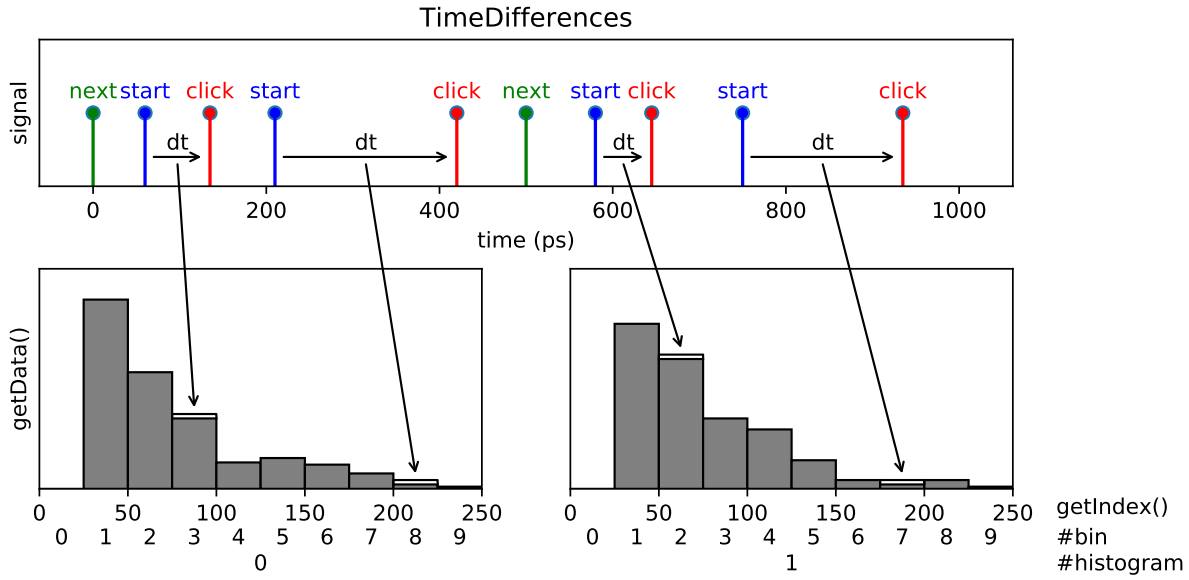
Return type 1D_array[float]

getIndex()

Returns A vector of size *n_bins* containing the time bins in ps.

Return type 1D_array[int]

TimeDifferences



A one-dimensional array of time-difference histograms with the option to include up to two 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 consecutive 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 time-difference data can be accumulated into a single histogram or into multiple subsequent histograms. In this way, you can record a sequence of time-difference histograms. To switch from one histogram to the next one you have to specify a channel that provide switch markers (*next_channel* parameter). Also you need to specify the number of histograms with the parameter *n_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 marker 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.

class TimeDifferences(*tagger*, *click_channel*, *start_channel=CHANNEL_UNUSED*, *next_channel=CHANNEL_UNUSED*, *sync_channel=CHANNEL_UNUSED*, *binwidth=1000*, *n_bins=1000*, *n_histograms=1*)

Parameters

- **tagger** (TimeTaggerBase) – 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

Note: A rollover occurs on a *next_channel* event while the histogram index is already in the last histogram. If *sync_channel* is defined, the measurement will pause at a rollover until a *sync_channel* event occurs and continues at the next *next_channel* event. With undefined *sync_channel*, the measurement will continue without interruption at histogram index 0.

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).

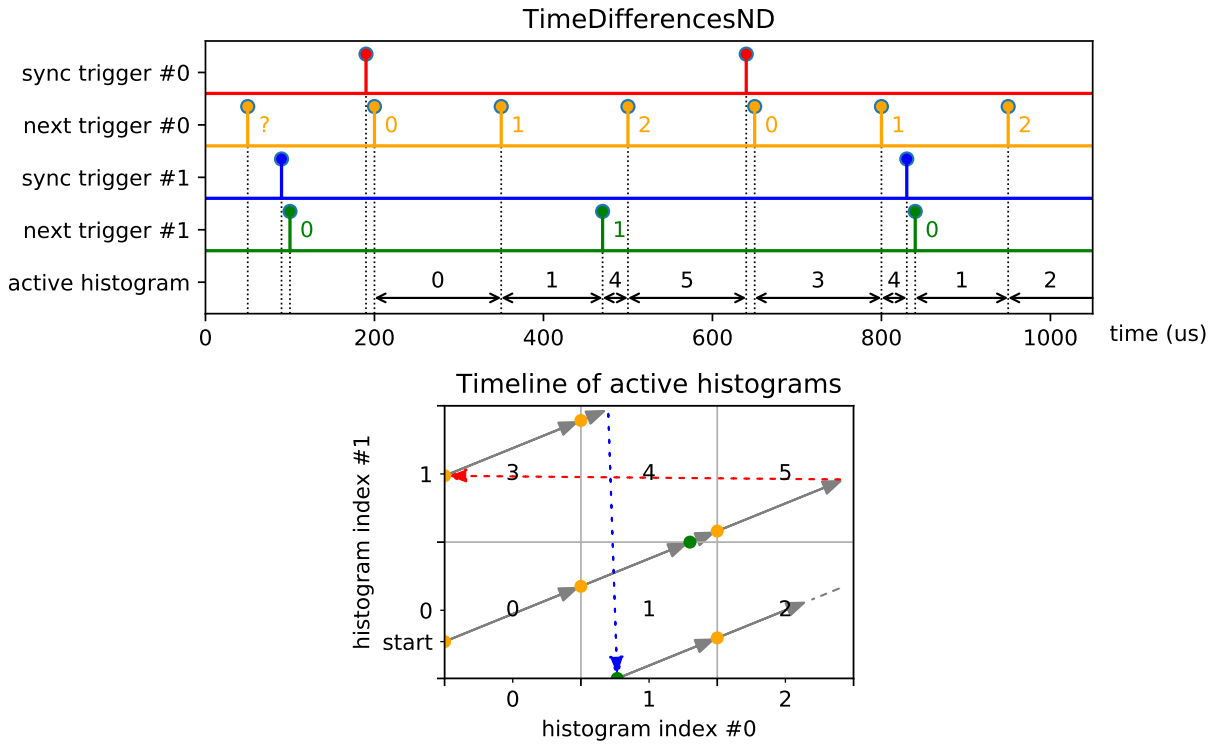
Return type int

ready()

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

Return type bool

TimeDifferencesND



This is an implementation of the `TimeDifferences` measurement class that extends histogram indexing into multiple dimensions.

Please read the documentation of `TimeDifferences` first. It captures many multiple start - multiple stop histograms, but with many asynchronous *next_channel* triggers. After each tag on each *next_channel*, the histogram index of the associated dimension is incremented by one and reset to zero after reaching the last valid index. The elements of the parameter *n_histograms* specify the number of histograms per dimension. The accumulation starts when *next_channel* has been triggered on all dimensions.

You should provide a synchronization trigger by specifying a sync_channel per dimension. It will stop the accumulation when an associated histogram index rollover occurs. A sync event will also stop the accumulation, reset the histogram index of the associated dimension, and a subsequent event on the corresponding *next_channel* starts the accumulation again. The synchronization is done asynchronous, so an event on the *next_channel* increases the histogram index even if the accumulation is stopped. The accumulation starts when a tag on the *sync_channel* arrives with a subsequent tag on *next_channel* for all dimensions.

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

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

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

Parameters

- **tagger** (TimeTaggerBase) – 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_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.

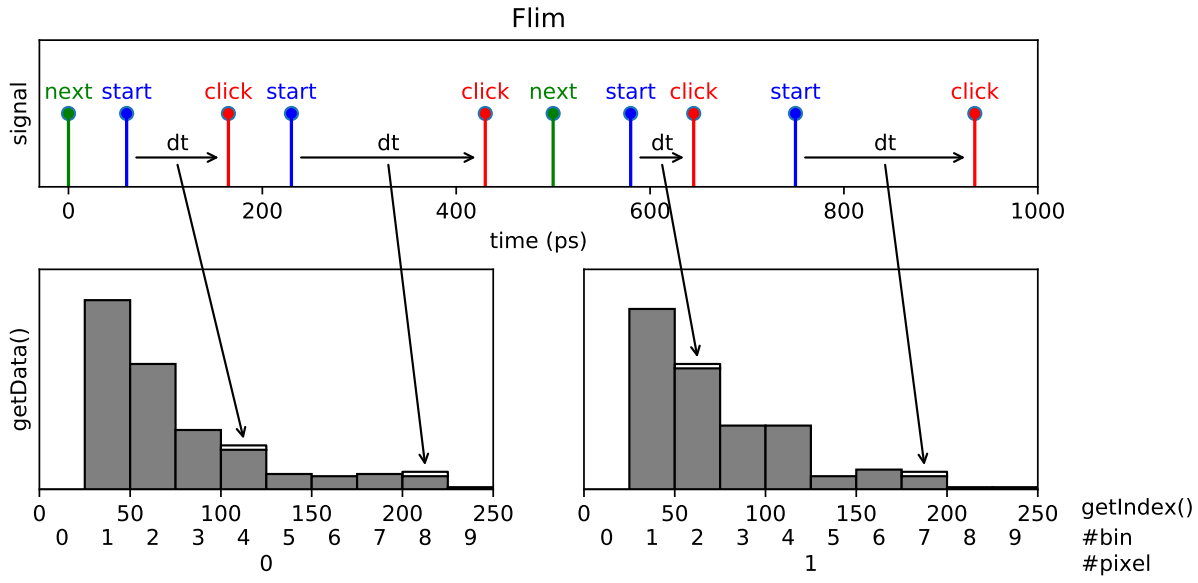
7.5.5 Fluorescence-lifetime imaging (FLIM)

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 not final yet. It has a very advanced functionality, but details are subject to change. Please give us feedback (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 before 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** (TimeTaggerBase) – time tagger object instance
- **start_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click_channel** (*int*) – channel on which start clicks are received for the time differences histogramming
- **pixel_begin_channel** (*int*) – start marker 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, 2D array with dimensions [n_bins, n_pixels].

Return type 2D_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 1D_array[float]

getFramesAcquired()

Returns The number of frames that have been completed so far, since the creation or 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. 2D array with dimensions [n_bins, n_pixels]

Return type 2D_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 1D_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 1D_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** (*1D_array[int]*) – 1D array containing the raw histogram data, with the data of pixel `i` and time bin `j` at index `i * n_bins + j`
- **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: **int**
number of pixels of the frame

bins: **int**
number of bins of each histogram

frame_number: **int**
current frame number

pixel_count: **int**
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, 2D array with dimensions [n_bins, n_pixels].

Return type 2D_array[int]

getIntensities()

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

Return type 1D_array[float]

getSummedCounts()

Returns The summed counts of each histogram.

Return type 1D_array[int]

getPixelBegins()

Returns An array of the start timestamps of each pixel.

Return type 1D_array[int]

getPixelEnds()

Returns An array of the end timestamps of each pixel.

Return type 1D_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** (*TimeTaggerBase*) – time tagger object instance
- **start_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click_channel** (*int*) – channel on which start clicks are received for the time differences histogramming
- **pixel_begin_channel** (*int*) – start marker 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** (*1D_array[int]*) – 1D array containing the raw histogram data, with the data of pixel `i` and time bin `j` at index `i * n_bins + j`
- **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.

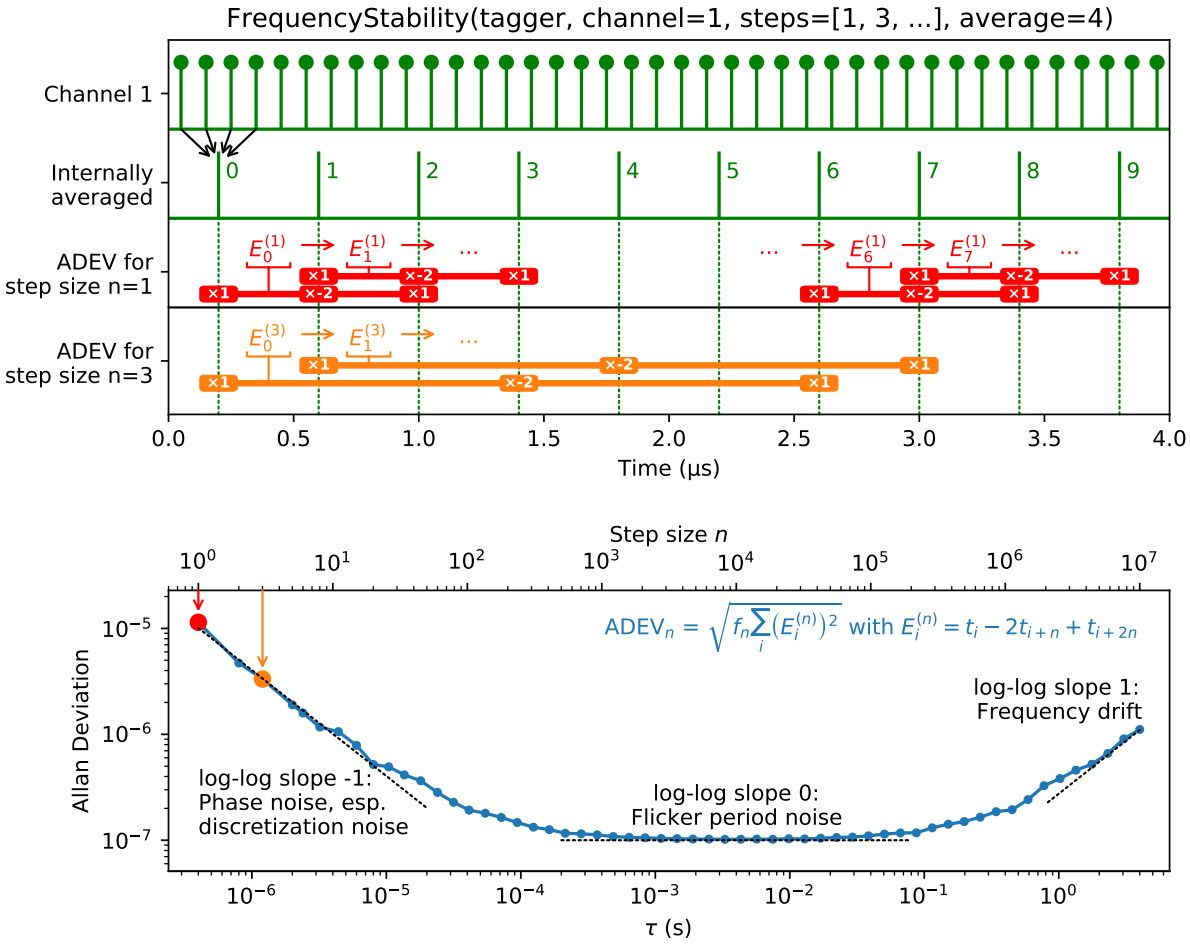
on_frame_end()

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

7.5.6 Frequency analysis

FrequencyStability

Frequency Stability Analysis is used to characterize periodic signals and to identify sources of deviations from the perfect periodicity. It can be employed to evaluate the frequency stability of oscillators, for example. A set of established metrics provides insights into the oscillator characteristics on different time scales. The most prominent metric is the Allan Deviation (ADEV). The *FrequencyStability* class executes the calculation of often used metrics in parallel and conforms to the IEEE 1139 standard. For more information, we recommend the Handbook of Frequency Stability Analysis.


FrequencyStability(tagger, channel=1, steps=[1, 3, ...], average=4)
Channel 1
Internally averaged
ADEV for step size n=1
ADEV for step size n=3
0 1 2 3 4 5 6 7 8 9
x1
x-2
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
Time (µs)
Step size n
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7
Allan Deviation
10^{-5} 10^{-6} 10^{-7}
$\mathrm{ADEV}_n = \sqrt{f_n \sum_i (E_i^{(n)})^2}$ with $E_i^{(n)} = t_i - 2t_{i+n} + t_{i+2n}$
log-log slope 1:
Frequency drift
log-log slope -1:
Phase noise, esp.
discretization noise
log-log slope 0:
Flicker period noise
10^{-6} 10^{-5} 10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^0
τ (s)

The calculated deviations are the root-mean-square $\sqrt{f_n \sum_i \left(E_i^{(n)}\right)^2}$ of a specific set of error samples $E^{(n)}$ with a normalization factor f_n. The step size n together with the oscillator period T defines the time span $\tau_n = nT$ that is investigated by the sample. The error samples $E^{(n)}$ are calculated from the phase samples t that are generated by the *FrequencyStability* class by averaging over the timestamps of a configurable number of time-tags. To investigate the averaged phase samples directly, a trace of configurable length is stored to display the current evolution of frequency and phase errors.

Each of the available deviations has its specific sample $E^{(n)}$. For example, the Allan Deviation investigates the second derivative of the phase t using the sample $E_i^{(n)} = t_i - 2t_{i+n} + t_{i+2n}$. The full formula of the Allan deviation for a set of N averaged timestamps is

$$\mathrm{ADEV}(\tau_n) = \sqrt{\frac{1}{2(N-2n)\tau_n^2} \sum_{i=1}^{N-2n} (t_i - 2t_{i+n} + t_{i+2n})^2}.$$

The deviations can be displayed in the Allan domain or in the time domain. For the time domain, the Allan domain data is multiplied by a factor proportional to τ. This means that in a log-log plot, all slopes of the time domain curves are increased by +1 compared to the Allan ones. The factor $\sqrt{3}$ for ADEV/MDEV and $\sqrt{10/3}$ for HDEV, respectively, is used so that the scaled deviations of a white phase noise distortion correspond to the standard deviation of the averaged timestamps t. In some cases, there are different established names for the representations. The *FrequencyStability* class provides numerous metrics for both domains:

Allan domain	Time domain
	Standard Deviation (STDD)
Allan Deviation (ADEV)	ADEVScaled = $\frac{\tau}{\sqrt{3}}$ ADEV
Modified Allan Deviation (MDEV)	Time Deviation TDEV = $\frac{\tau}{\sqrt{3}}$ MDEV
Hadamard Deviation (HDEV)	HDEVScaled = $\frac{\tau}{\sqrt{10/3}}$ HDEV

class FrequencyStability(*tagger*, *channel*, *steps*, *average*, *trace_len*)

Parameters

- **channel** (*int*) – The input channel number.
- **steps** (*list[int]*) – The step sizes to consider in the calculation. The length of the list determines the maximum number of data points. Because the oscillator frequency is unknown, it is not possible to define τ directly.
- **average** (*int*) – The number of time-tags to average internally. This downsampling allows for a reduction of noise and memory requirements. Default is 1000.
- **trace_len** (*int*) – Number of data points in the phase and frequency error traces, calculated from averaged data. The trace always contains the latest data. Default is 1000.

Note: Use *average* and *TimeTagger.setEventDivider()* with care: The event divider can be used to save USB bandwidth. If possible, transfer more data via USB and use *average* to improve your results.

getDataObject()

Returns An object that allows access to the current metrics

Return type *FrequencyStabilityData*

class FrequencyStabilityData

getTau()
The τ axis for all deviations. This is the product of the *steps* parameter of the *FrequencyStability* measurement and the measured average period of the signal.

Returns The τ values.

Return type 1D_array[float]

getADEV()
The overlapping Allan deviation, the most common analysis framework. In a log-log plot, the slope allows one to identify the type of noise:

- -1: white or flicker phase noise like discretization or analog noisy delay
- -0.5: white period noise
- 0: flicker period noise like electric noisy oscillator
- 0.5: integrated white period noise (random walk period)
- 1: frequency drift, e.g., induced thermally

Sample $E_i^{(n)} = t_i - 2t_{i+n} + t_{i+2n}$

Domain Allan domain

Returns The overlapping Allan Deviation.

Return type 1D_array[float]

getMDEV()
Modified overlapping Allan deviation. It averages the second derivate before calculating the RMS. This splits the slope of white and flicker phase noise:

- -1.5: white phase noise, like discretization
- -1.0: flicker phase noise, like an electric noisy delay

The metric is more commonly used in the time domain, see *getTDEV()*.

Sample $E_i^{(n)} = \frac{1}{n}\sum_{j=0}^{n-1}\left(t_{i+j} - 2t_{i+j+n} + t_{i+j+2n}\right)$

Domain Allan domain

Returns The overlapping Modified Allan Deviation.

Return type 1D_array[float]

getHDEV()
The overlapping Hadamard deviation uses the third derivate of the phase. This cancels the effect of a constant phase drift and converges for more divergent noise sources at higher slopes:

- 1: integrated flicker period noise (flicker walk period)
- 1.5: double integrated white period noise (random run period)

It is scaled to match the ADEV for white period noise.

Sample $E_i^{(n)} = t_i - 3t_{i+n} + 3t_{i+2n} - t_{i+3n}$

Domain Allan domain

Returns The overlapping Hadamard Deviation.

Return type 1D_array[float]

getSTDD()

> **Caution:** The standard deviation is not recommended as a measure of frequency stability because it is non-convergent for some types of noise commonly found in frequency sources, most noticeable the frequency drift.

Standard deviation of the periods.

Sample $E_i^{(n)} = t_i - t_{i+n} - \mathrm{mean}_k(t_k - t_{k+n})$

Domain Time domain

Returns The standard deviation.

Return type 1D_array[float]

getADEVScaled()

Domain Time domain

Returns The scaled version of the overlapping Allan Deviation, equivalent to *getADEV()* * *getTau()* / $\sqrt{3}$.

Return type 1D_array[float]

getTDEV()

The Time Deviation (TDEV) is the common representation of the Modified overlapping Allan deviation *getMDEV()*. Taking the log-log slope +1 and the splitting of the slope of white and flicker phase noise into account, it allows an easy identification of the two contributions:

- -0.5: white phase noise, like discretization
- 0: flicker phase noise, like an electric noisy delay

Domain Time domain

Returns The overlapping Time Deviation, equivalent to *getMDEV()* * *getTau()* / $\sqrt{3}$.

Return type 1D_array[float]

getHDEVScaled()

> **Caution:** While HDEV is scaled to match ADEV for white period noise, this function is scaled to match the TDEV for white phase noise. The difference of period vs phase matching is roughly 5% and easy to overlook.

Domain Time domain

Returns The scaled version of the overlapping Hadamard Deviation, equivalent to *getHDEV()* * *getTau()* / $\sqrt{10/3}$.

Return type 1D_array[float]

getTraceIndex()

The time axis for *getTracePhase()* and *getTraceFrequency()*.

Returns The time index in seconds of the phase and frequency error trace.

Return type 1D_array[float]

getTracePhase()
Provides the time offset of the averaged timestamps from a linear fit over the last *trace_len* averaged timestamps.

Returns A trace of the last *trace_len* phase samples in seconds.

Return type 1D_array[float]

getTraceFrequency()
Provides the frequency offset from the average frequency during the last *trace_len + 1* averaged timestamps.

Returns A trace of the last *trace_len* normalized frequency error data points in pp1.

Return type 1D_array[float]

7.5.7 Time-tag-streaming

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

Time tag format

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

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

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

Allows user to access a copy of the time tag stream. It allocates a memory buffer of the size *max_tags* which is filled with the incoming time tags that arrive from the specified channels. User shall call `getData()` method periodically to obtain the current buffer containing timetags collected. This action will return the current buffer object and create another empty buffer to be filled until the next call to `getData()`.

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

Parameters

- **tagger** (`TimeTaggerBase`) – time tagger object instance
- **n_max_events** (`int`) – buffer size for storing time tags
- **channels** (`list[int]`) – non-empty list of channels to be captured.

See all common methods

getData()

Returns a `TimeTagStreamBuffer` object and clears the internal buffer of the `TimeTagStream` measurement. Clearing the internal buffer on each call to `getData()` guarantees that consecutive calls to this method will return every time-tag only once. Data loss may occur if `getData()` is not called frequently enough with respect to *n_max_events*.

Returns buffer object containing timetags collected.

Return type *TimeTagStreamBuffer*

class TimeTagStreamBuffer

hasOverflows: bool

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

tStart: int

Return the data-stream time position when the `TimeTagStream` or `FileWriter` started data acquisition.

tGetData: int

Return the data-stream time position of the call to *.getData()* method that created this object.

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: please use `getEventTypes()` instead.

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]

FileWriter

Writes the time-tag-stream into a file in a structured binary format with a lossless compression. The estimated file size requirements are 2-4 Bytes per time tag, not including the container the data is stored in. The continuous background data rate for the container can be modified via *TimeTagger.setStreamBlockSize()*. Data is processed in blocks and each block header has a size of 160 Bytes. The default processing latency is 20 ms, which means that a block is written every 20 ms resulting in a background data rate of 8 kB/s. By increasing the processing latency via *setStreamBlockSize(max_events=524288, max_latency=1000)* to 1 s, the resulting data rate for the container is reduced to one 160 B/s. The files created with *FileWriter* measurement can be read using *FileReader* or loaded into the Virtual Time Tagger.

Note: You can use the *Dump* for dumping into a simple uncompressed binary format. However, you will not be able to use this file with Virtual Time Tagger or *FileReader*.

The *FileWriter* is able to split the data into multiple files 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 *TimeTaggerBase.getConfiguration()* method. The configuration is always written into every file.

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

Parameters

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

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

Example:

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

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

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

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

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

class FileReader(*filenames*)

This is the class 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 (dictionary in Python) that contains the Time Tagger configuration at the time of file creation.

Return type str or dict

Dump

Deprecated since version 2.6.0: please use *FileWriter* instead.

Warning: The files created with this class cannot be read by *TimeTaggerVirtual* and *FileReader*.

Writes the timetag stream into a file in a simple uncompressed binary format that store timetags as 128bit records, see *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** (TimeTaggerBase) – 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)

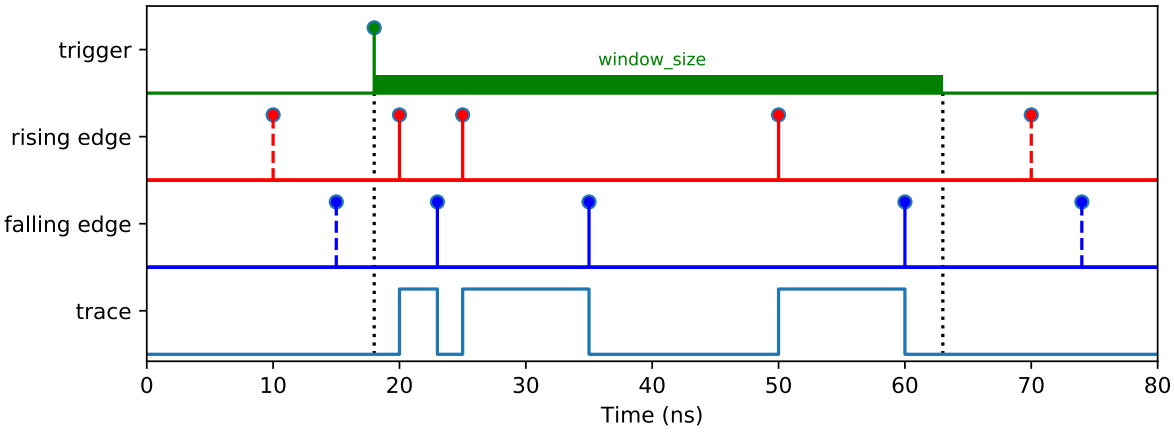
clear()

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

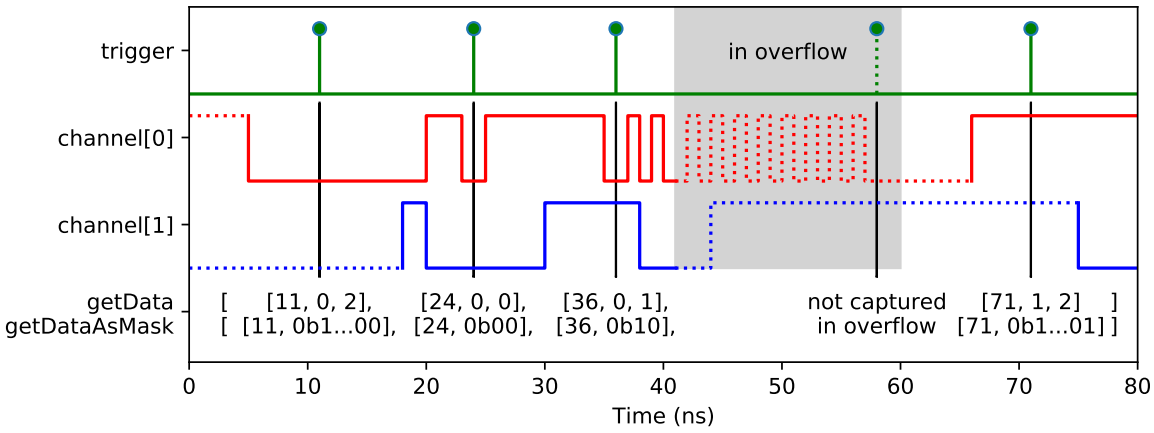
Current input state. Can be unknown because no edge has been detected on the given channel after initialization or an overflow.

UNKNOWN

HIGH

LOW

Sampler



The *Sampler* class allows sampling the state of a set of channels via a trigger channel.

For every event on the trigger input, the current state (low: 0, high: 1, unknown: 2) will be written to an internal buffer. Fetching the data of the internal buffer will clear its internal buffer. So every event will be returned only once.

Time Tagger detects pulse edges and therefore a channel will be in the unknown state until an edge detection event was received on that channel from the start of the measurement or after an overflow. The internal processing assumes that no event could be received within the channel's deadtime otherwise invalid data will be reported until the next event on this input channel.

The maximum number of channels is limited to 63 for one *Sampler* instance.

class Sampler(*tagger*, *trigger*, *channels*, *max_trigger*)

Parameters

- **tagger** (*TimeTagger*) – TimeTagger object
- **trigger** (*int*) – Channel number of the trigger signal
- **channels** (*list[int]*) – List of channels to be sampled
- **max_trigger** (*int*) – The number of triggers and their respective sampled data, which is stored within the measurement class.

See all common methods

getData()

Returns and removes the stored data as a 2D array (n_triggers x (1+n_channels)):

```
[
    [timestamp of first trigger,  state of channel 0, state of channel 1, ...
↪],
    [timestamp of second trigger, state of channel 0, state of channel 1, ...
↪],
    ...
]
```

Where state means:

```
0: low
1: high
2: undefined (after overflow)
```

Returns sampled data

Return type 2D_array[int]

getDataAsMask()

Returns and removes the stored data as a 2D array (n_triggers x 2):

```
[
    [timestamp of first trigger,  (state of channel 0) << 0 | (state of␣
↪channel 1) << 1 | ... | any_undefined << 63],
    [timestamp of second trigger, (state of channel 0) << 0 | (state of␣
↪channel 1) << 1 | ... | any_undefined << 63],
    ...
]
```

Where state means:

```
0: low or undefined (after overflow)
1: high
```

If the highest bit (data[63]) is marked, one of the channels has been in an undefined state.

Returns sampled data

Return type 2D_array[int]

7.5.8 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** (TimeTaggerBase) – 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.

7.5.9 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: This functionality is only available for C++, C# and Python. You can find examples of how to use the *CustomMeasurement* in your examples folder.

---

class CustomMeasurement(*tagger*)

Parameters **tagger** (TimeTaggerBase) – 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.

See all common methods

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.array() 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']
```

---

mutex

Context manager object (see Context Manager Types) that locks the mutex when used and automatically unlocks it when the code block exits. For example, it is intended for use with Python's "with" keyword as

```
class MyMeasurement(CustomMeasurement):

    def getData(self):
        # Acquire a lock for this instance to guarantee that
        # self.data is not modified in other parallel threads.
        # This ensures to return a consistent data.
        with self.mutex:
            return self.data.copy()
```

CHAPTER

EIGHT

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.

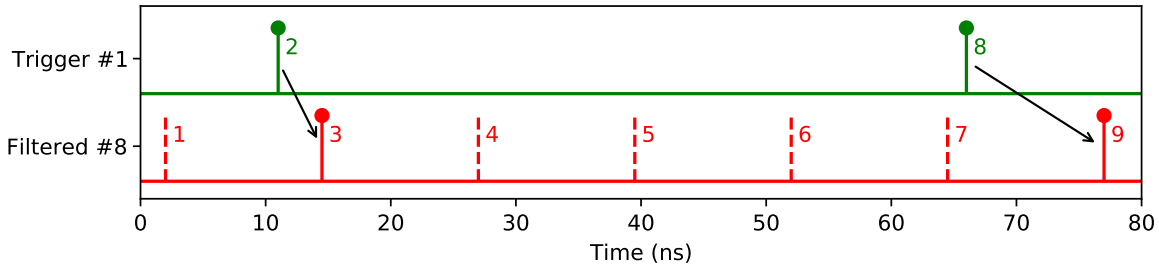
8.1.1 Example configurations

One trigger and one filtered channel

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

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

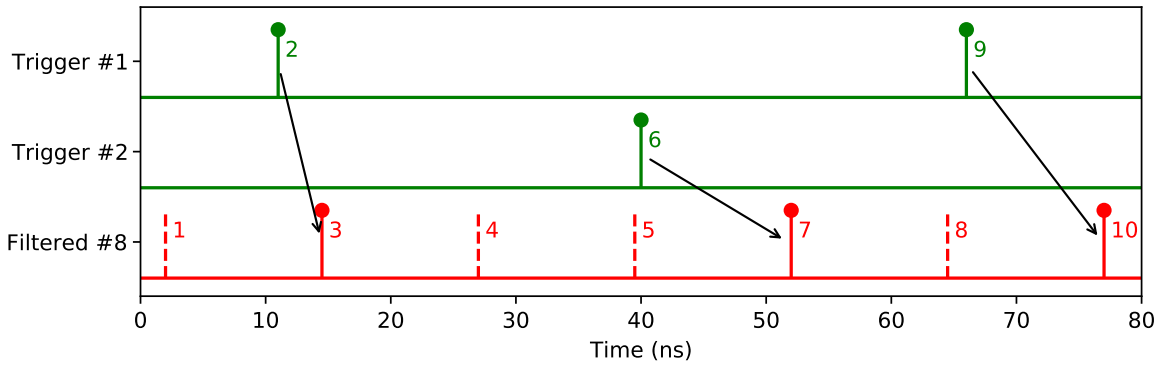
The Conditional Filter discards by default all signals of the filtered-channel. Only the very next event is transmitted after an event on the trigger-channel. In the example, click 2 opens the gate for click 3. When click 3 passes, it closes the gate and the subsequent events will be discarded until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

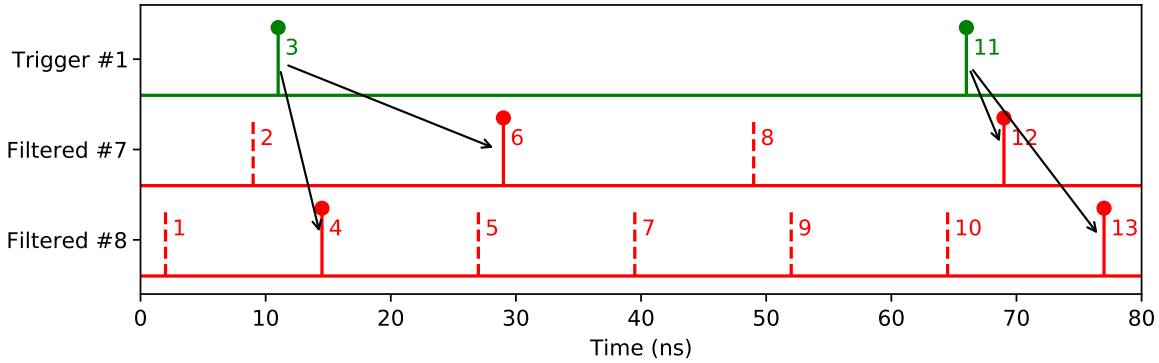


This is the typical use case when you detect photons with multiple detectors and want to correlate both with the common excitation laser.

Multiple filtered channels

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

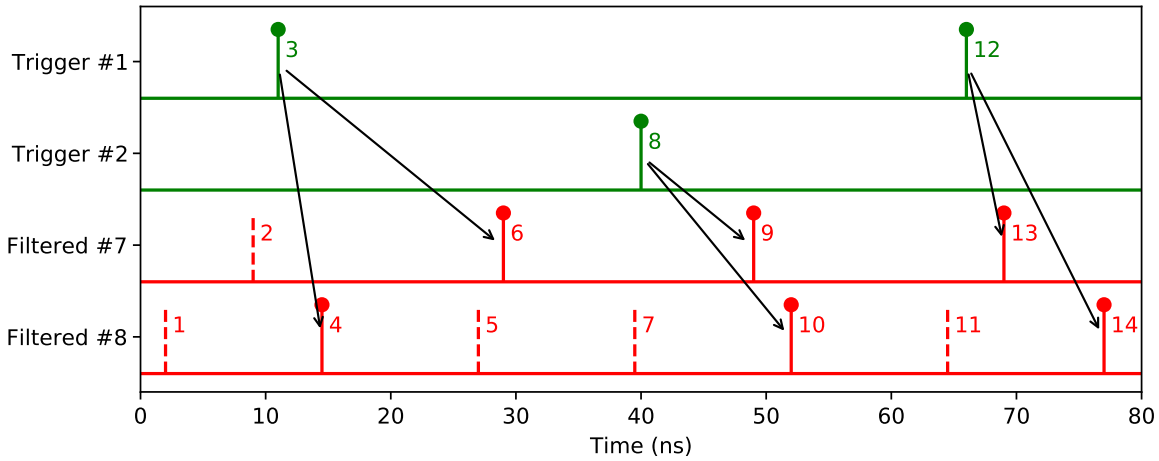
```
tagger.setConditionalFilter(trigger=[1], filtered=[7, 8])
```



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 `TimeTaggerBase.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 `TimeTaggerBase.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 `TimeTaggerBase.setDelaySoftware()`.
8. **Delayed Channel:** The most flexible way to control the relative delay of your signals are Virtual Channels.

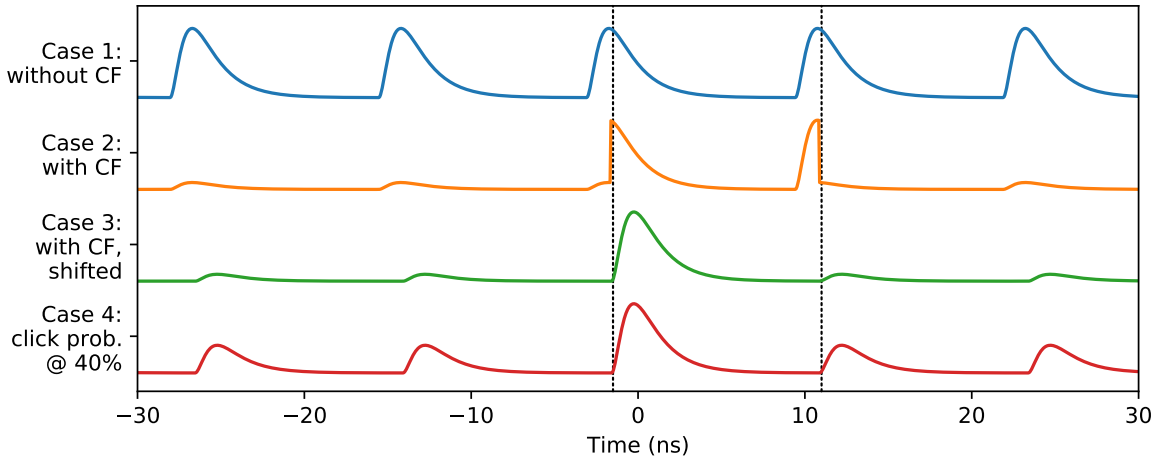
Consequences

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)
```

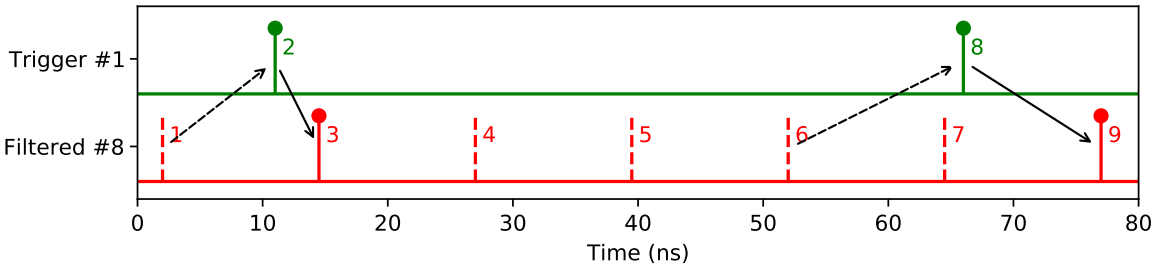
Case 1: Without the Conditional Filter set up, the Correlation measurement class provides a periodic signal. The periodicity is a result of the multi-start/multi-stop approach of the Correlation measurement: A click on the detector will contribute together with any laser synchronization pulse to the 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 *`TimeTaggerBase.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

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 *TimeTaggerBase.getFence()* and then wait for it to be signaled with *waitForFence()* at any time later. If you want to create a fence and immediately wait for it then using the *sync()* method is more convenient.

CHAPTER

NINE

LINUX

9.1 Supported distributions

- CentOS 8
- Ubuntu 18.04 LTS
- Ubuntu 20.04 LTS

will be discontinued within 2021

- CentOS 7
- Ubuntu 16.04 LTS

9.2 Installation

Download and install the package for your linux distribution from the Time Tagger downloads page https://www.swabianinstruments.com/time-tagger/downloads/.

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.

9.3 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 may cause an error message to appear.

9.4 Time Tagger with Python

Supported Python versions are 2.7, 3.5, 3.6, 3.7, 3.8, 3.9.

- 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.

9.5 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`).

9.6 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 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="`python3-config --includes --libs`"
NUMPY_FLAGS="-I`python3 -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++ -shared _TimeTagger.cxx $CFLAGS -o _TimeTagger.so
```

CHAPTER

TEN

FREQUENTLY ASKED QUESTIONS

10.1 How to detect falling edges of a pulse?

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

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

10.2 What value should I pass to an optional channel?

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

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

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

10.4 How do I choose a binwidth for a histogram?

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

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

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

   ```
   histogram_span = binwidth * n_bins
   ```

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

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

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

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

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

CHAPTER

ELEVEN

USAGE STATISTICS COLLECTION

You can help us developing and improving the Time Tagger by enabling automated usage statistics collection. The usage statistics data collection is designed to help us better understand how the Time Tagger hardware and software are used. This data includes the performance indicators, configuration, the state of the Time Tagger, and API usage patterns. The usage statistics data is pseudonymized[1] and cannot be linked to a specific user or specific hardware unit. On installation of the Time Tagger software, a random *user_id* will be created and added to the usage statistics reports. Users can review the contents of usage statistics data by using the `getUsageStatisticsReport()`. Also users can disable usage statistics data collection at any time via Time Tagger API as `setUsageStatisticsStatus(UsageStatisticsStatus.Disabled)`. It is possible to enable the usage statistics collection temporarily and without automatic uploading which may be helpful for debugging.

11.1 Contents of the usage statistics data

- Internal calibration data.
- Hardware sensor data obtainable with `TimeTagger.getSensorData()` but with the serial number obscured.
- Time Tagger's configuration as returned by `getConfiguration()` but with the serial number obscured.
- All warning and error messages produced by the Time Tagger software. All identifying information like serial numbers is obscured.
- Average, minimal, and maximal aggregate data rate sent over USB in each usage session.
- Usage and configuration of the measurements and their performance indicators.
- Computer's processor name and capabilities, as well as the RAM size.

11.2 Ways of control

You will be asked to join Time Tagger improvement program during software installation. You can change your decision at any later time by uninstalling and installing the Time Tagger software again.

You can also control usage statistics collection through the programming interface. The following examples show how to perform key operations of enabling, disabling, and retrieving the usage statistics data. See also *Usage statistics functions*.

[1] Here "pseudonymized" means that the user retains privacy of their data and remain unidentified as long as their *user_id* (pseudonym) is not matched to their personal identity.

Get and set usage statistics collection status

```
# 0 - UsageStatisticsStatus.Disabled
# 1 - UsageStatisticsStatus.Collecting
# 2 - UsageStatisticsStatus.CollectingAndUploading
status = getUsageStatisticsStatus()

# Enable usage statistics collection without uploading
setUsageStatisticsStatus(UsageStatisticsStatus.Collecting)

# Enable usage statistics collection with uploading
setUsageStatisticsStatus(UsageStatisticsStatus.CollectingAndUploading)

# Disable usage statistics collection
setUsageStatisticsStatus(UsageStatisticsStatus.Disabled)
```

Get current usage statistics data

```
json_string = getUsageStatisticsReport()
```

CHAPTER

TWELVE

REVISION HISTORY

12.1 V2.10.4 - 23.02.2022

WebApp

- Fixed Input Delay for negative values.
- Fixed adding new channels for *Countrate* measurement.
- Fixed *HistogramLogBins* for low start times (< 10ps).
- Fixed units for data export of *Counter*, *Correlation*, *Histogram2D*, and *HistogramLogBins* measurements.

12.2 V2.10.2 - 31.12.2021

Improvements

- Added support for Python 3.10.

Fixes for issues since v2.10.0

- Fixed *HistogramLogBins* in the Web Application.
- Fixed *DelayedChannel* for negative delays.
- Fixed an issue with *Counter.getDataTotalCounts()* not resetting to 0 on *clear()*.
- Fixed some Matlab examples not being compatible with 2016b or older.

12.3 V2.10.0 - 22.12.2021

Highlights

- Time Tagger Network: All Time Tagger devices and the acquired data can be accessed via the network from multiple clients or locally across the different programming languages. The clients can use all TimeTagger measurement classes and may optionally control the settings of the physical Time Tagger.
- A new frequency stability toolbox: It offers on-the-fly evaluation of periodic signals by calculating several analysis metrics, including, for example, the Allan deviation (ADEV) and time deviation (TDEV).

- Software Clock: The new recommended method for using an external clock on the Time Tagger Ultra. The time tag stream is rescaled on the software side with respect to the connected clock. It allows for a broad input frequency range and also calculates phase error estimators. In addition, the input jitter of the clock channel will be averaged out, resulting in a lower jitter for measurements including the clock channel directly.

Features

- Counter: New *Counter.getDataObject()* returning data as an object of *CounterData* and allowing for continuous chunkwise data acquisition. This object contains the Counter data, timing information and overflow flags.
- New *HistogramND* measurement, which is a multidimensional generalization of the older *Histogram2D*.
- New *Sampler* measurement class for a triggered sampling of the current state of other channels.
- Measurement and virtual channel settings can now be requested with *getConfiguration()* method. The settings of all measurements are also available in the return value of *getConfiguration()* method.

WebApp

- A *Time Tagger Network* server can be activated in the settings.
- Includes the *Software Clock* feature.
- Adds *Event Divider* settings.
- Shows specified RMS jitter for each channel in HighRes mode.
- It is now possible to specify the integration time in a single-shot or cyclic mode (internally uses *startFor()*) for all available measurement classes.

Performance

- Improved performance of *Counter*, *Countrate*, *TimeTagStream*, *Combiner*, *DelayedChannel* for many channels.
- *HistogramLogBins* with an improved algorithm, multithreading, and AVX2+AVX512 tuning.
- *Coincidences* improved for high input rates with low coincidence rates.

Behavior change

- *TimeTagStream* now always requires a list of channels.
- *CustomMeasurement* in Python: `with self.mutex` replaces `self.lock` and `self.unlock`.
- A Synchronizer with only one Time Tagger will use the timestamps of the Synchronizer but the channel identifiers of the single device itself.
- No messages on the INFO level will be shown in Matlab to avoid running into deadlocks.
- `std::invalid_argument` exceptions are now wrapped as `ValueError` in Python.

Examples

- New Python example to measure the maximum transfer rate and the jitter.
- New Python example to show coincidence counting applications.
- New example to show the use of the software clock and measure the frequency stability of the test signal in Python, Matlab and LabVIEW.
- Update the *Counter* example in Python and Matlab to show the use of the new `CounterData`.

Fixes

- Skips an unlikely blocking `freeTimeTagger()` call for up to 10 seconds.
- Fixes the 64-bit signed integer overflow after 106 days on Linux.
- Stops playing the last sound of `setSoundFrequency()` after `freeTimeTagger()`.
- Fixes the timing of `TimeTagStreamBuffer.tGetData` in the last block of `FileReader`.
- Adds support for TTU HW revision 1.6 and TT20 Value.
- Fixes the empty configuration and channel list in `FileReader` before fetching the first time tag.
- Fixes a race condition on the Time Tagger Ultra, which may yield one invalid time tag after USB connection errors.
- Fixes a crash on using `with CustomMeasurement() as c` in Python.
- Fixes incorrectly displayed units in the WebApp if measurement settings changed during a measurement.
- Fixes the behavior of Histogram2D if start_channel matches a stop channel.
- Fixes the behavior of Countrate with startFor if it ends within an overflow interval.

12.4 V2.9.0 - 07.06.2021

Highlights

- Reduced communication latency of all Time Taggers.
- Reduced Time Tagger 20 crosstalk on channel 1 and 2.
- Improved USB connection stability for Time Tagger 20.
- Optional collection and reporting of pseudonymous usage statistics. *Improvement program*.
- Please use at least v2.9.0 for devices shipped from 2021 on.

Changes

- *TimeTaggerBase.getConfiguration()* and *TimeTagger.getSensorData()* return a JSON string with partially renamed sensor names.
- Altered *Countrate.getData()* to return `NaN` (Not a Number) for zero capture durations.
- Uses `enum.Enum` as base class for all enumerators in the Python wrapper (Python >= 3.5).
- Improved the format of the Time Tagger error messages.

Features

- Added *TimeTagger.setHardwareBufferSize()* for the Time Tagger 20.
- Added an example and tutorial on how to work with a remote Time Tagger using Python and the Pyro5 package.
- License upgrades can be flashed now for the Time Tagger 20 via the web application.

Bug fixes

- Fixed *TimeTagger.setStreamBlockSize()* block size heuristic while uploading new configuration.
- Fixed slow performance of *freeTimeTagger()* in overflow mode.
- Fixed *waitUntilFinished()* invoke nodes in LabVIEW examples.
- Fixed error message in the Web Application for non compatible devices.
- Fixed *TimeTaggerVirtual.getConfiguration()*. Now it is returning configuration data for *TimeTaggerVirtual* class.
- Fixed a possible crash on Python interpreter exit while running *CustomMeasurement*.
- Fixed *TimeTaggerBase.sync()* signaling one block too late. The fix reduces the sync, measurement start and clear times.

12.5 V2.8.4 - 04.05.2021

- Fixed the initialization for a Virtual Time Tagger in the Web Application

12.6 V2.8.2 - 26.04.2021

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

12.7 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.

Note: The release is fully compatible with all Time Tagger 20 devices. It is compatible with all Time Tagger Ultra devices shipped from March 2021 and all earlier Time Tagger Ultra devices with 8 or less channels without HighRes option. If you received Time Tagger Ultra before March 2021 and it has more than 8 channels or HighRes, it is not compatible with the release. Please contact support to get a free device exchange to be fully compatible again.

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 *TimeTaggerBase.setInputDelay()* into *TimeTaggerBase.setDelayHardware()* and *TimeTaggerBase.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.
- Maximum input frequency of the Time Tagger Ultra is reduced to 475 MHz.
- The deadtime specification of the Time Tagger Ultra changed to 2.1 ns. It can detect events separated by 2 ns with possible loss of some events.

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.

12.8 V2.7.6 - 26.04.2021

- Fixed RuntimeError "Got the USB error 'UnsupportedFeature'" when calling *createTimeTagger()*

12.9 V2.7.4 - 19.04.2021

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

12.10 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 *TimeTaggerBase.waitForFence()* in Matlab.
- Fixed *TimeTaggerBase.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.

12.11 V2.7.0 - 01.10.2020

Highlights

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

Fixes and improvements

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

12.12 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.

12.13 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.

12.14 V2.6.6 - 10.07.2020

Highlights

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

Webapp

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

Time Tagger Ultra

- Hardware revision 1.1 now with the same performance enhancement of 500 MHz maximum sync rate, 2ns dead time and better phase stability, as introduced before for Hardware revision > 1.1
- Dropped support for the very first Time Tagger Ultras, an error will be shown on initialization - free exchange program available
- More intuitive byte order of the bitmask in setLED
- Small modifications to the hardware channel to channel delay

Backend

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

12.15 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

12.16 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

12.17 V2.6.0 - 23.12.2019

Highlights

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

Known issues

- FileWriter and FileReader have a low performance on network devices

API changes

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

New features

- Web App: Normalization (counts/s) for the Counter measurement
- getConfiguration returns the current hardware configuration as a JSON string
- added g2 normalization for HistogramLogBins with getDataNormalizedG2
- improved overflow behavior for Countrate due to the missed event counters
- improved overflow handling for the g2 normalization of Correlation and HistogramLogBin
- support for Python version 3.8
- smaller latency on low data rates due to adaptive chunk sizes of <= 20 ms
- support for the Time Tagger Ultra hardware revision 1.4

Examples

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

Bug fixes

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

12.18 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

12.19 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

12.20 V2.4.0 - 10.04.2019

Libraries

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

API

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

Examples

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

Time Tagger Ultra

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

Time Tagger 20

- 166.6 MHz EXT input clock detection enabled

Operating systems

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

12.21 V2.2.4 - 29.01.2019

- fix the conditional filter with filter and trigger events arriving within one clock cycle
- fix issue with negative 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

12.22 V2.2.2 - 13.11.2018

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

12.23 V2.2.0 - 07.11.2018

General improvements

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

API

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

Web application

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

12.24 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

12.25 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

12.26 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

12.27 V2.1.0 - 06.03.2018

Time Tagger Ultra

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

12.28 V2.0.4 - 01.02.2018

Bug fixes

- Closing the web application server window works properly now

12.29 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.

12.30 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

12.31 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

12.32 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

12.33 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

12.34 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

12.35 V1.0.0

initial release supporting python

12.36 Channel Number Schema 0 and 1

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

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

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

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

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

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

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

INDEX

G

H

I

L

M

O

P

R

S

T

U

W