9.15  V1.0.6 - 16.03.2017 ................................................. 57
9.16  V1.0.4 - 24.11.2016 ................................................. 58
9.17  V1.0.2 - 28.07.2016 ................................................. 58
9.18  V1.0.0 ................................................................. 58
9.19  Channel Number Schema 0 and 1 ............................... 58
The following section describes how to get started with your Time Tagger.

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

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

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

In addition, information about the hardware, API, etc. can be found in the menu bar on the left and on our main website: http://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 (GUI) from the Windows start menu
3. the web application should show up in your browser

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

1. click Add TimeTagger, click create
2. click Add measurement, look for Correlation and click Add next to it
3. select Rising edge 1 for Channel 1 and Rising edge 2 for Channel 2
4. set binwidth to 10 ps and leave n_bins at 1000, click initialize

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

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

Where to go from here…

To learn more about the Time Tagger web application you are encouraged to consult the following resources.

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

1.2 Python

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

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

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

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

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

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

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

1.3 LabVIEW (via .NET)

A set of examples is provided in .\examples\LabVIEW for LabVIEW 2014 and higher (32 and 64 bit).
1.4 Matlab (wrapper for .NET)

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

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

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

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

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

1.5 Wolfram Mathematica (via .NET)

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

1.6 .NET

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

The following are important to note:

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

1.7 C#

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

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

1.8 C++

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

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

2.1.1 Operating System

Windows Windows 7 or higher
We provide separate Windows installers for 32 and 64 bit systems.

2.1.2 Installation

Download and install the most recent Time Tagger software from our downloads site.
Connect the Time Tagger to your computer with the USB cable.
You should now be ready to use your Time Tagger.

2.1.3 Web Application

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

2.1.4 Programming Examples

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

HARDWARE

3.1 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.

3.1.1 Electrical characteristics

<table>
<thead>
<tr>
<th>Property</th>
<th>Time Tagger 20</th>
<th>Time Tagger Ultra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>50 Ohm</td>
<td>50 Ohm</td>
</tr>
<tr>
<td>Input voltage range</td>
<td>0.0 to 5.0 V</td>
<td>-5.0 to 5.0 V</td>
</tr>
<tr>
<td>Trigger level range</td>
<td>0.0 to 2.5 V</td>
<td>-2.5 to 2.5 V</td>
</tr>
<tr>
<td>Minimum signal level</td>
<td>100 mV</td>
<td>100 mV</td>
</tr>
<tr>
<td>Minimum pulse width</td>
<td>1.2 ns</td>
<td>0.8 ns</td>
</tr>
</tbody>
</table>

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

3.3 Status LEDs

The *Time Tagger* has two LEDs showing status information. A green LED turns on when the USB power is connected. An RGB LED shows the information tabulated below.
3.4 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.

3.5 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).

3.6 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.

3.7 Synthetic dead time

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

3.8 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 reduced time tag stream is processed by the software in the exact same fashion as the full time tag stream.

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

The time resolution of the filter is the very same as the dead time of the channels (Time Tagger 20: 6 ns, Time Tagger Ultra: 2.25 ns).
To ensure deterministic filter logic, the physical time difference between the filtered channels and triggered channels must be larger than $\pm ( \text{deadtime} + 3 \text{ ns})$. The Conditional Filter works also in the regime when signals arrive almost simultaneously, but one has to be aware of a few details described below. Note also that software-defined input delays as set by the method `setInputDelay()` do not apply to the Conditional Filter logic.

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

### 3.9 Bin equilibration

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.

### 3.10 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.

### 3.11 General purpose IO (available upon request)

The device is ready to be equipped with up to four general purpose IO ports (SMA-connectors), and an external clock input or output. These can be used to implement custom features such as special fast input or output triggers, enable / disable gates, software controllable input and output lines, and so on. Please contact us for custom designs.

### 3.12 External Clock Input

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

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

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

CLK Input requirements:

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

- Time Tagger Ultra
  - 100 mVpp - 3 Vpp AC coupled into 50 Ohm, 500 mVpp recommended

Performance:

The input clock signal must be stable very low jitter signal to achieve the best performance to reach the specified performance without an external clock input. 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:

To reach the low specified input jitter for the Time Tagger with an external clock, the input signals must be uncorrelated to the external clock.
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.

4.1 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.

4.2 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.

4.3 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.
5.1 Overview

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

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

5.1.1 Examples

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

**Measuring cross-correlation**

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

```python
# 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)

# Wait until the settings above are applied
ntagger.sync()
```

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

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

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

## Using virtual channels

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

tagger = createTimeTagger()

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

# Wait until the settings above are applied
tagger.sync()

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

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

# Wait and print the countrate all three channels
pause(1)
print(rate.getData())
>> [ 800008.81 800008.81 1600017.62]

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

## Using multiple Time Taggers

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

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

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

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

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

Analog voltage levels are specified in Volts.

5.1.3 Channel numbers

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

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

5.1.4 Unused channels

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

5.2 Module constants

CHANNELUNUSED

Can be used instead of a channel number when no specific channel is assumed.

5.3 Module functions

createTimeTagger(serial="")

Establishes the connection to a first available Time Tagger device and creates a TimeTagger object. Optionally, the connection to a specific device can be achieved by specifying the device serial number.

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

Returns TimeTagger object

Return type TimeTagger

freeTimeTagger(tagger)

Releases all Time Tagger resources and terminates the active connection.

Parameters tagger (TimeTagger) – TimeTagger object to disconnect

scanTimeTagger()

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

Returns List of serial numbers

Return type List(string)
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().

g timevalTaggerChannelNumberScheme ()
Returns the currently used channel schema which is either TT_CHANNEL_NUMBER_SCHEME_ZERO or TT_CHANNEL_NUMBER_SCHEME_ONE.

Returns Channel schema
Return type int32

5.4 The TimeTagger class

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

class TimeTagger

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

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

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

g etTriggerLevel (channel)
Returns trigger level for the specified physical channel number.

Parameters channel (int32) – Physical channel number

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

Return type double

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

Parameters
- channel (int32) – Channel number
- delay (int64) – Delay time in picoseconds
**getInputDelay** *(channel)*
Get the input delay compensation for the given channel in picoseconds.

- **Parameters**
  - channel *(int32)* – Channel number
- **Returns**
  - Delay time in picoseconds
- **Return type**
  - int64

**setConditionalFilter** *(trigger, filtered)*
Activates or deactivates the event filter. Time tags on the filtered channels are discarded unless they were preceded by a time tag on one of the trigger channels which reduces the data rate. More details can be found in the In Depth Guide: Conditional Filter.

- **Parameters**
  - trigger *(list[int32])* – List of channel numbers
  - filtered *(list[int32])* – List of channel numbers

**getConditionalFilterTrigger** *
Returns the collection of trigger channels for the conditional filter.

- **Returns**
  - List of channel numbers
- **Return type**
  - list[int32]

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

- **Returns**
  - List of channel numbers
- **Return type**
  - list[int32]

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

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

**getEventDivider** *(channel)*
Gets the event divider filter factor for the given channel.

- **Parameters**
  - channel *(int32)* – Channel number
- **Returns**
  - Divider factor value
- **Return type**
  - uint32

**setNormalization** *(state)*
Enables or disables Gaussian normalization of the detection jitter. Enabled by default.

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

---

5.4. The TimeTagger class
getNormalization()  
Returns true if Gaussian normalization is enabled.  

Returns True/False  
Return type bool

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

Parameters  
• channel (int32) – Channel number.
• deadtime (int64) – Deadtime value in picoseconds.

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

Return type int64

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

Parameters channel (int32) – Physical channel number  

Returns Deadtime value in picoseconds  
Return type int64

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

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

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

Parameters channel (int32) – Physical channel number

Returns True/False

Return type bool

getSerial()  
Returns the hardware serial number.

Returns Serial number string

Return type string

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

Returns Number of overflows

Return type int64
getOverflowsAndClear()  
Returns the number of overflows that occurred since start-up and sets them to zero (see, clearOverflows()).

Returns  Number of overflows
Return type  int64

clearOverflows()  
Set the overflow counter to zero.

clearOverflows()  
Set the overflow counter to zero.

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

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

Returns  PCB version
Return type  string

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

Returns  Min and max voltage in Volt
Return type  (double, double)

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

Parameters  channel (int32) – Channel number

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

Parameters  channel (int32) – Channel number

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

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

Parameters  type (int) – Defines what channels to be returned
Returns  List of channel numbers
Return type  list[int32]

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

Parameters  channel (int32) – Channel number

Returns  Channel number
Return type  int32

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

Parameters  channel (int32) – Channel number

Returns  True/False

Return type  bool

setHardwareBufferSize(size) - TT Ultra only
Sets the maximum buffer size within the Time Tagger Ultra. The default value is 40 MTags, but can be
changed within the range of 32 kTags to 512 MTags. Please note that this buffer can only be filled with a
total data rate of up to 500 MTags/s.

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

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

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

Return type  list[double]

getDistributionCount()
Returns the calibration data represented in counts.

Returns  Distribution data

Return type  2d_array[int64]

getDistributionPSec()
Returns the calibration data in picoseconds.

Returns  Calibration data

Return type  2d_array[int64]

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

Returns  Clock period in picoseconds

Return type  int64

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

Parameters  divider (int32) – Division factor

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

Returns  Tabulated sensor data

Return type  string

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

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

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LED off</td>
</tr>
<tr>
<td>1</td>
<td>LED on</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal LED behavior, not overwritten by setLED</td>
</tr>
<tr>
<td>1</td>
<td>LED state is overwritten by the corresponding bit of 0-8</td>
</tr>
</tbody>
</table>

**mask bits**

- 16-18: status, rgb - all Time Tagger models
- 19-21: power, rgb - Time Tagger Ultra only
- 22-24: clock, rgb - Time Tagger Ultra only

**Parameters**

- `bitmask (uint32)` – LED bitmask.

**setFilter (state)**

Depreciated since version V1.0.3: use `setConditionalFilter()` instead.

**getFilter ()**

Depreciated since version V1.0.3: use `getConditionalFilterTrigger()` and `getConditionalFilterFiltered()` instead.

**getChannels ()**

Depreciated since version V1.2.0: use `getChannelList ()` instead.

## 5.5 Virtual Channels

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

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

### 5.5.1 Available virtual channels

- **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 an input channel 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.

### 5.5.2 Common methods

VirtualChannel.getChannel()
VirtualChannel.getChannels()

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

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

### 5.5.3 Combiner

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

```python
class Combiner(tagger, channels=[1])
```

**Parameters**

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

Detected 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)

**Parameters**

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

5.5.5 Coincidences

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

Example code:

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

# or equivalent but less performant

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

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

---

5.5. Virtual Channels
class Coincidences (tagger, coincidenceGroups, coincidenceWindow)

Parameters

• tagger (TimeTagger) – time tagger object instance
• coincidenceGroups (list[list[int32]]) – list of channel groups on which coincidence will be detected in the virtual channel
• coincidenceWindow (int64) – maximum time between all events for a coincidence [ps]

5.5.6 FrequencyMultiplier

Frequency Multiplier for a channel with a periodic signal.

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

class FrequencyMultiplier (tagger, input_channel, multiplier)

Parameters

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

5.5.7 GatedChannel
Transmits the signal from an input channel to a new virtual channel between an edge detected at the gate_start_channel and the gate_stop_channel.

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

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

### 5.5.8 DelayedChannel

Clones an input channel, which can be delayed by a time specified with the delay parameter in the constructor or the setDelay() method.

**Note:** If you want to set a global delay for an input channel, it is recommended to use TimeTagger. setInputDelay() as more computationally efficient, instead of creating a virtual DelayedChannel.

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

**Parameters**
- **tagger** *(TimeTagger)* – time tagger object
- **input_channel** *(int32)* – channel which is delayed
- **delay** *(int64)* – amount of time to delay in ps, must be positive

**setDelay**(delay)

Allows modifying the delay time.

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

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

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

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

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

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

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

5.6 Measurement Classes

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

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

5.6.1 Available measurement classes

**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.

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

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

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

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

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

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

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

SynchronizedMeasurements  Helper class that allows synchronization of the measurement classes.

Dump  This 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.

5.6.2 Common methods

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

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

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

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

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

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

Note: All measurements start accumulating data immediately after their creation.
**Returns** True/False

**Return type** bool

`getCaptureDuration()`

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

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

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

The specific measurements are described below.

### 5.6.3 Event counting

**Countrate**

![Countrate](image)

\[
\text{getData()} = \frac{\text{counts}}{\text{time}} \\
= \frac{7}{800\text{ns}} \\
= 8750000 \text{ counts/second}
\]

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

**class Countrate** *(tagger, channels)*

**Parameters**

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

**getData()**

Returns the average count rate in counts per second.

**getCountsTotal()**

Returns the total amount of events since the instantiation of this object.

**clear()**

Resets the accumulated counts to zero and restarts the measurement with the next incoming tag.
Counter

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

```python
class Counter(tagger, channels, binwidth, n_values)

Parameters

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

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

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

`clear`()
Resets the array to zero and restarts the measurement.
```
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 \( \text{begin\_channel} \). When a tag is detected on the \( \text{begin\_channel} \) it starts counting tags on the \( \text{click\_channel} \). When the next tag is detected on the \( \text{begin\_channel} \) 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 \( \text{end\_channel} \) is specified, the measurement stores the current counter value and resets the counter when a tag is detected on the \( \text{end\_channel} \) rather than the \( \text{begin\_channel} \). You can use this, e.g., to accumulate counts within a gate by using rising edges on one channel as the \( \text{begin\_channel} \) and falling edges on the same channel as the \( \text{end\_channel} \). The accumulation time for each value can be accessed via \( \text{getBinWidths()} \). The measurement stops when all entries in the data vector are filled.

**class** **CountBetweenMarkers** *(tagger, click\_channel, begin\_channel, end\_channel, n\_values)*

**Parameters**

- \( \text{tagger} \) *(TimeTagger)* – time tagger object
- \( \text{click\_channel} \) *(int32)* – channel on which clicks are received, gated by \( \text{begin\_channel} \) and \( \text{end\_channel} \)
- \( \text{begin\_channel} \) *(int32)* – channel that triggers the beginning of counting and stepping to the next value
- \( \text{end\_channel} \) *(int32)* – channel that triggers the end of counting
- \( \text{n\_values} \) *(uint32)* – number of values stored (data buffer size)

**get\_Data()**

Returns an array of size \( n_{values} \) containing the acquired counter values.

**getIndex()**

Returns a vector of size \( n_{values} \) containing the time in ps of each start click in respect to the very first start click.

**getBinWidths()**

Returns a vector of size \( n_{values} \) containing the time differences of ‘start -> (next start or stop)’ for the acquired counter values.

**clear()**

Resets the array to zero and restarts the measurement.
ready

Returns True when the entire array is filled.

5.6.4 Time histograms

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

StartStop

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

class StartStop(tagger, click_channel, start_channel, binwidth)

Parameters

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

getData()

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

clear()

Resets the array to zero and restarts the measurement.
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.

```csharp
class Histogram (tagger, click_channel, start_channel, binwidth, n_bins)

Parameters

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

**get_data**

Returns a one-dimensional array of size `n_bins` containing the histogram.

**get_index**

Returns a vector of size `n_bins` containing the time bins in ps.

**clear**

Resets the array to zero.
```

**HistogramLogBins**

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

Counts per bin

Counts per bin / binwidth

class HistogramLogBins
Parameters

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

**get_data**
Returns a one-dimensional array of size \( n\_bins \) containing the histogram.

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

**get_bin_edges**
Returns a vector of size \( n\_bins + 1 \) containing the bin edges in picoseconds.

**clear**
Resets the array to zero.

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

**class Correlation**(tagger, channel\_1, channel\_2, binwidth, n\_bins)

**Parameters**
- **tagger**(TimeTagger) – time tagger object
- **channel\_1**(int32) – reference channel
- **channel\_2**(int32) – second channel (when left empty or set to CHANNEL\_UNUSED -> an auto-correlation measurement is performed, which is the same as setting channel\_1 = channel\_2)
- **binwidth**(uint64) – bin width in ps
- **n\_bins**(uint) – the number of bins in the resulting histogram

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

**getDataNormalized()**
Return the data normalized as:

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

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

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

**clear()**
Resets the accumulated data.
5.6.5 Advanced time histograms

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

**FLIM**

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

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

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

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

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

```java
class FLIM(tagger, click_channel, start_channel, next_channel, binwidth, n_bins, n_pixels)
```

**Parameters**

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

\texttt{get\_Data()}

Returns a two-dimensional array of size \textit{n\_bins} by \textit{n\_pixels} containing the histograms.

\texttt{get\_Index()}

Returns a vector of size \textit{n\_bins} containing the time bins in ps.

\texttt{clear()}

Resets the array to zero.

**TimeDifferences**

A multidimensional histogram measurement with the option up to include three additional channels that control how to step through the indices of the histogram array. This is a very powerful and generic measurement. You can use it to record cross-correlation, lifetime measurements, fluorescence lifetime imaging and many more measurements based on pulsed excitation. Specifically, the measurement waits for a tag on the \textit{start\_channel}, then measures the time difference between the start tag and all subsequent tags on the \textit{click\_channel} and stores them in a histogram. If no \textit{start\_channel} is specified, the \textit{click\_channel} is used as \textit{start\_channel} corresponding to an auto-correlation measurement. The histogram has a number \textit{n\_bins} of bins of bin width \textit{binwidth}. Clicks that fall outside the histogram range are discarded. Data accumulation is performed independently for all start tags. This type of measurement is frequently referred to as ‘multiple start, multiple stop’ measurement and corresponds to a full auto- or cross-correlation measurement.

The data obtained from subsequent start tags can be accumulated into the same histogram (one-dimensional measurement) or into different histograms (two-dimensional measurement). In this way, you can perform more general two-dimensional time-difference measurements. The parameter \textit{n\_histograms} specifies the number of histograms. After each tag on the \textit{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 \textit{next\_channel}.

You can also provide a synchronization trigger that resets the histogram index by specifying a \textit{sync\_channel}. The measurement starts when a tag on the \textit{sync\_channel} arrives with a subsequent tag on \textit{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 \textit{next\_channel} starts the accumulation again.
Typically, you will run the measurement indefinitely until stopped by the user. However, it is also possible to specify the maximum number of rollovers of the histogram index. In this case, the measurement stops when the number of rollovers has reached the specified value. This means that for both a one-dimensional and for a two-dimensional measurement, it will measure until the measurement went through the specified number of rollovers / sync tags.

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

**Parameters**

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

**getTimeData()**

Returns a two-dimensional array of size `n_bins` by `n_histograms` containing the histograms.

**getTimes()**

Returns a vector of size `n_bins` containing the time bins in ps.

**clear()**

Resets all data to zero.

**setMaxCounts()**

Sets the number of rollovers at which the measurement stops integrating.

**getCounts()**

Returns the number of rollovers (histogram index resets).

**getStatus()**

Returns ‘true’ when the required number of rollovers set by `setMaxCounts()` has been reached.

**Overflow handling**

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

- **sync_channel and next_channel are both defined** the measurement stops integrating at an overflow and continues with the next signal on the sync_channel
- **only next_channel is defined** the histogram index is reset at the overflow and the next signal on the next_channel starts the integration again
- **sync_channel and next_channel are both undefined** the accumulation continues

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

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

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

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

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

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

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

Parameters

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

### 5.6.6 Timetag streaming

Measurement classes described in this section provide direct access to the time tag stream with minimal pre-processing.

**Dump**

Writes the timetag stream into a file in a binary format.

Each data block transferred to the PC is stored in a file and has the following format:

<table>
<thead>
<tr>
<th>Size</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bit</td>
<td>int8</td>
<td>overflow (0 == false or 1 == true)</td>
</tr>
<tr>
<td>24 bit</td>
<td>-</td>
<td>reserved</td>
</tr>
<tr>
<td>32 bit</td>
<td>int32</td>
<td>channel number</td>
</tr>
<tr>
<td>64 bit</td>
<td>int64</td>
<td>time in ps from device start-up</td>
</tr>
</tbody>
</table>

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

```python
class Dump (tagger, filename, max_tags, channels)

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

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

**stop**()
Stops data recording and closes data file.
```

**TimeTagStream**

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

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

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

class TimeTagStreamBuffer

**getTimestamps**
Returns an array of timestamps.

**Returns** Event timestamps in picoseconds for all chosen channels.
**Return type** list(int64)

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

**Returns** Channel number for each detected event.
**Return type** list(int64)

**getOverflows**(self)
Returns an array of overflow flags for every timestamp.

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

**Returns** True/False
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.

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

Parameters

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

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

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

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

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

5.6.7 Helper classes

SynchronizedMeasurements

\textit{(beta version)}

The \textit{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.

\textbf{Warning:} Currently, it is not possible to synchronize the \textit{Histogram} and \textit{FLIM} measurements. If you need to synchronize these measurements, please use \textit{TimeDifferences} for their equivalent and synchronizable implementations, as shown below.

\textbf{Equivalency between \textit{Histogram} and \textit{TimeDifferences}:}

\begin{verbatim}
Histogram(tagger, click_channel=1, start_channel=2, 
        binwidth=100, n_bins=1000)
# Histogram using TimeDifferences
TimeDifferences(tagger, click_channel=1, start_channel=2, 
               next_channel=CHANNEL UNUSED, sync_channel=CHANNELUNUSED, 
               binwidth=100, n_bins=1000, n_histograms=1)
\end{verbatim}

\textbf{Equivalency between \textit{FLIM} and \textit{TimeDifferences}:}

\begin{verbatim}
FLIM(tagger, click_channel=1, start_channel=2, next_channel=3, 
     binwidth=100, n_bins=1000, n_pixels=320*240)
# FLIM using TimeDifferences
TimeDifferences(tagger, click_channel=1, start_channel=2, 
                next_channel=3, sync_channel=CHANNELUNUSED, 
                binwidth=100, n_bins=1000, n_histograms=320*240)
\end{verbatim}

class SynchronizedMeasurements (tagger)

\textbf{Parameters}  \textit{tagger} (\textit{TimeTagger}) – TimeTagger object

\textbf{registerMeasurement} (\textit{measurement})

Registers the \textit{measurement} object into a pool of the synchronized measurements.

\textbf{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 \textit{SynchronizedMeasurements} object after registering the measurement objects which should be synchronized.

\textbf{Parameters}  \textit{measurement} – Any measurement (Iterator) object.
**start()**
Calls `start()` for every registered method in a synchronized way.

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

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

**clear()**
Calls `clear()` for every registered method in a synchronized way.
6.1 Conditional Filter

The Conditional Filter allows you to decrease the time tag rate without losing those time tags that are relevant to your application. 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.

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

This feature is enabled by the Conditional Filter. 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 transmission of a subsequent tag on the filtered channels.

```cpp
setConditionalFilter(<vector int> trigger, <vector int> filtered)
```

**Caution:** The time resolution of the Conditional Filter is the very same as the dead time of the channels (Time Tagger 20: 6 ns, Time Tagger Ultra: 2.25 ns). To ensure deterministic filter logic, the physical time difference between the filtered channels and triggered channels must be larger than +/- (deadtime + 3 ns). The Conditional Filter works also in the regime when signals arrive almost simultaneously, but one has to be aware of a few details described below. Note also that software defined input delays as set by the method `setInputDelay()` do not apply to the Conditional Filter logic.

6.2 Detailed description if the Conditional Filter

For signals attached to the Time Tagger which exceed the USB transfer limit, the Conditional Filter can be enabled to reduce the transmitted data to the PC.

```cpp
setConditionalFilter(<vector int> trigger, <vector int> filtered)
```

The total maximum for the average data transfer rate is about 8-9 million events per second for the Time Tagger 20 without the Conditional Filter and for the Time Tagger Ultra about 65 million tags depending on the CPU of the PC and the complexity of the evaluation of the signal. Let’s assume a high frequency 50 MHz laser sync is applied to channel 7, further called #7, and a single photon detector (APD) is attached to #0. As soon as a correlation measurement between these two channels is initiated,
the Time Tagger world ran into data overflows without the filter applied. Overflows are indicated by a red LED or can be retrieved by calling `tagger.getOverflows()`. The correlation measurement would still work but the effective correlation rate drops by a factor of \( \frac{\text{total input rate}}{\text{max transfer rate}} \).

Often, the data rate of the APD channel is much lower compared to the sync channel and a lot of sync signals can be discarded without losing information for the correlation measurement. Therefore, the Conditional Filter can be applied to reduce the data rate:

**Case a)** absolute time difference between the trigger-channel and the filtered-channel \( \geq (\text{deadtime} + 3 \text{ ns}) \) -> deterministic signal

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:

```
setConditionalFilter(trigger=[0], filtered=[7])
```

The event 1 of #7 is not transmitted because there has been no event at #0 up to that point. Only event 3 of #7 is transmitted because it succeeds the event 2 of #0. In principle, event 1 and 2 belong together, but since the signal applied to filtered-channel is usually periodic, the correlation of pulse 3 with 2 is the same as the correlation of 1 with 2 except for a constant time delay depending on the period of the filtered channel.

To have event 2 and 3 on top of each other for the evaluation (e.g. correlation measurement), `setInputDelay(0, dt)` can be applied.

If it is required that the corresponding event (event 1 of the example) is transmitted, the signal of the filtered-channel must be delayed with a cable such that event 1 arrives after the event 2 with at least a time difference of 3 ns. The delay of typical coax cable is 5 ns/m.

### 6.2.1 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:

```
setConditionalFilter(trigger=[0,1], filtered=[7])
```
Because of event 2 event 3 is transmitted, and because of event 4 event 5 is transmitted.

### 6.2.2 Multiple filtered channels

Another option is to use the filter with one trigger-channel and several filtered-channels:

```python
setConditionalFilter(trigger=[0], filtered=[6,7])
```

All the very next events of the filtered-channels will be transmitted (event 5 of #7 and event 7 of #6) after an event at the trigger-channel (event 4).

**Case b)** Time difference between the triggered-channel and the filtered-channel < (deadtime + 3 ns) -> deterministic signal & non-deterministic signal

For the following consideration, a constant shift of the input signal (up to 3ns) between the channels is neglected. If the events of the trigger-channel and the filtered-channels are very close together, as for case a), it is guaranteed that the very next event on the filtered channel will be transmitted after an event recognized in the trigger-channel. But in addition, the event after might be transmitted in addition.
or the event before, depending on the order the signals arrive.

Conclusions and remarks: There can be only one Conditional Filter enabled at a time. Shift the signals with cables (5 ns/m) such that that the signals arrive with a time gap of at least (deadtime + 3ns) to work in the deterministic regime (Case a)). When possible, start using the Time Tagger without the Conditional Filter and accept overflows. Compare to what happens when the Conditional Filter is turned on. setInputDelay() does not shift the channels in time for the Conditional Filter. That means setInputDelay has no effect on the Conditional Filter. This input delay is applied in software after the signals have been transmitted to the PC. If it is necessary to delay a signal for the Conditional Filter, the cable length needs to be adjusted. Countrate/Counter measurements on the filteredChannels will show the filtered count rate instead of the rate of the input signal setEventDivider() is an alternative method to reduce high data rates

6.2.3 Disable the Conditional Filter

The Conditional Filter can be disabled by passing empty vectors to the setConditionalFilter method:

```csharp
setConditionalFilter([], [])
```
The package installs the Python and C++ libraries for amd64 systems including example programs.

**Graphical user interface (web application):**

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

**Known issues**

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

**Using the Time Tagger with Python 2.7 or 3:**

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

**Using the Time Tagger with C++:**

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

**General remark:**

- Please contact us in case you have any questions or comments about the Ubuntu or CentOS package and/or the API for the Time Tagger.
FREQUENTLY ASKED QUESTIONS

8.1 How to detect falling edges of a pulse?

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

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

8.2 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`.

8.3 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.

8.4 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?

   \[ \text{histogram span} = \text{binwidth} \times n\_\text{bins} \]

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

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

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

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

   Smaller binwidth will require a longer time to accumulate the sufficient number of counts to achieve desired noise level compared to larger binwidth. This is referring to a shot-noise that is proportional to \( 1/\sqrt{N} \) where \( N \) is a number of counts in a single bin. This is the very same concept as SNR improvement by averaging. Larger binwidths will naturally get larger counts per bin in a shorter time for the same signal rates.
9.1 V2.4.4 - 19.07.2019

- reduced crosstalk between nonadjacent channels.
- improved and more detailed documentation.
- new method `Countsrate.getCountsTotal()` that returns number of events.
- 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` virtual channel that sometimes resulted in no data flow.

9.2 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

9.3 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

9.4 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

9.5 V2.2.2 - 13.11.2018
• Removed not required Microsoft prerequisites.
• 32 bit version available

9.6 V2.2.0 - 07.11.2018

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

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

Web application
• full ‘setConditionalFilter’ functionality available from the backend within the Web application

9.7 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

9.8 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

9.9 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

9.10 V2.1.0 - 06.03.2018

Time Tagger Ultra
• efficient buffering of up to 60 MTags within the device to avoid overflows
9.11 V2.0.4 - 01.02.2018

Bug fixes

- Closing the web application server window works properly now

9.12 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.

9.13 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

9.14 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 (beta)
- `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

### 9.15 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
9.16 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


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

9.18 V1.0.0

initial release supporting python

9.19 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.

<table>
<thead>
<tr>
<th>Time Tagger 20 and Ultra 8</th>
<th>Time Tagger Ultra 18</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>rising 0 to 7</td>
<td>rising 0 to 17</td>
<td>TT_CHANNEL_NUMBER_SCHEME_ZERO</td>
</tr>
<tr>
<td>falling 8 to 15</td>
<td>falling 18 to 35</td>
<td></td>
</tr>
<tr>
<td>new 1 to 8</td>
<td>new -1 to -18</td>
<td>TT_CHANNEL_NUMBER_SCHEME_ONE</td>
</tr>
</tbody>
</table>

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.
autoCalibration() TimeTagger method, 20

CHANNEL_UNUSED built-in variable, 15
clear(), 27
clear() Correlation method, 34
clear() CountBetweenMarkers method, 30
clear() Counter method, 29
clear() CountRate method, 28
clear() Dump method, 39
clear() FLIM method, 36
clear() Histogram method, 32
clear() HistogramLogBins method, 33
clear() StartStop method, 31
clear() SynchronizedMeasurements method, 43
clear() TimeDifferences method, 37
clearOverflows() TimeTagger method, 19
Coincidence built-in class, 23
Coincidences built-in class, 23
Combiner built-in class, 22
ConstantFractionDiscriminator built-in class, 26
Correlation built-in class, 34
CountBetweenMarkers built-in class, 30
Counter built-in class, 29
CountRate built-in class, 28
createTimeTagger() built-in function, 15

DelayedChannel built-in class, 25
Dump built-in class, 39

FLIM built-in class, 35
freeTimeTagger() built-in function, 15
FrequencyMultiplier built-in class, 24

GatedChannel built-in class, 25
getBinEdges() HistogramLogBins method, 33
getBinWidths() CountBetweenMarkers method, 30
getCaptureDuration(), 28
getchannel() VirtualChannel method, 22
getchannelList() TimeTagger method, 19
getchannels() TimeTagger method, 21
getchannels() TimeTagStreamBuffer method, 40
getchannels() VirtualChannel method, 22
getConditionalFilterFiltered() TimeTagger method, 17

getConditionalFilterTrigger() TimeTagger method, 17
clear() TimeDifferences method, 37
clearTotal() CountRate method, 28
data() , 27, 41
data() Correlation method, 34
data() CountBetweenMarkers method, 30
data() Counter method, 29
data() CountRate method, 28
data() FLIM method, 36
data() Histogram method, 32
data() HistogramLogBins method, 33
data() StartStop method, 31
data() TimeDifferences method, 37
data() TimeTagStream method, 40
dataNormalized() Correlation method, 34
dataNormalized() HistogramLogBins method, 33
dataDeadtime() TimeTagger method, 18
dataDistributionCount() TimeTagger method, 20
dataDistributionPsec() TimeTagger method, 20
dataEventDivider() TimeTagger method, 17
dataFilter() TimeTagger method, 21
dataIndex() Correlation method, 34
dataIndex() CountBetweenMarkers method, 30
dataIndex() Counter method, 29
dataIndex() FLIM method, 36
dataIndex() Histogram method, 32
dataIndex() TimeDifferences method, 37
dataInputDelay() TimeTagger method, 16
dataInvertedChannel() TimeTagger method, 19
getData() TimeTagger method, 18
dataOverflows() TimeTagStreamBuffer method, 40
dataOverflowsAndClear() TimeTagger method, 18
dataPcbVersion() TimeTagger method, 19
dataPsPerClock() TimeTagger method, 20
dataSerial() TimeTagger method, 18
dataTestSignal() TimeTagger method, 18
dataTimestamps() TimeTagStreamBuffer method, 40
dataTimeTaggerChannelNumberScheme() built-in function, 16
dataTriggerLevel() TimeTagger method, 16
hasOverflows() TimeTagStreamBuffer method, 40
Histogram built-in class, 32
HistogramLogBins built-in class, 33

isChannelUnused() TimeTagger method, 20
isRunning(), 27

ready() CountBetweenMarkers method, 30
ready() TimeDifferences method, 37
registerChannel() TimeTagger method, 19
registerMeasurement() SynchronizedMeasurements method, 42
reset() TimeTagger method, 16

scanTimeTagger() built-in function, 15
Scope built-in class, 41
setConditionalFilter() TimeTagger method, 17
setDeadtime() TimeTagger method, 18
setDelay() DelayedChannel method, 25
setEventDivider() TimeTagger method, 17
setFilter() TimeTagger method, 21
setInputDelay() TimeTagger method, 16
setLED() TimeTagger method, 20
setMaxCounts() TimeDifferences method, 37
setNormalization() TimeTagger method, 17
setTestSignal() TimeTagger method, 18
setTestSignalDivider() TimeTagger method, 20
setTimeTaggerChannelNumberScheme() built-in function, 15
setTriggerLevel() TimeTagger method, 16
start(), 27
start() SynchronizedMeasurements method, 42
startFor(), 27
startFor() SynchronizedMeasurements method, 43
StartStop built-in class, 31
stop(), 27
stop() Dump method, 39
stop() SynchronizedMeasurements method, 43
clear() TimeTagger method, 19
SynchronizedMeasurements built-in class, 42

TimeDifferences built-in class, 37
TimeDifferencesND built-in class, 38
TimeTagger built-in class, 16
TimeTagStream built-in class, 39
TimeTagStreamBuffer built-in class, 40

unregisterChannel() TimeTagger method, 19