Time Tagger User Manual

Release 2.3.5

Swabian Instruments

Feb 21, 2019

CONTENTS

CHAPTER

ONE

GETTING STARTED

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

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

- 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://timetagger.info

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 learnt 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

A simple correlation measurement is provided in `.\examples\LabVIEW\` for LabVIEW 2016 and higher. The requirements for Using .NET assemblies in LabVIEW can be found can be found here:

Note: Depending on whether 32 or 64 bit LabVIEW is installed, include the corresponding 32 or 64 bit .NET library to your project

1.4 Matlab

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

1.5 DLL/wrappers for .NET (e.g. Matlab, LabVIEW)

We provide a .NET class library (32 and 64 bit) for the TimeTagger which can be used to access the TimeTagger from high-level languages `.\driver\dotNet\TTCSharpxx.dll`.

The following are important to note:

- Namespace: `SwabianInstruments.TimeTagger`
- static functions (e.g. to create an instance of a TimeTagger) are accessible via `SwabianInstruments.TimeTagger.TT`

1.6 C#

A sample project how to use the .NET class library can 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.

1.7 C++

The provided Visual Studio 2017 C++ project can be found in `.\examples\cpp\`. Using the C++ interface is the most performant way to interact with the TimeTagger but more elaborate compared to the other high-level languages. Please visit `.\examples\cpp\TimeTagger-api.pdf` for more details on the C++ API.

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

2.1 Requirements

2.1.1 Operating System

Windows Windows 7 or higher

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

2.1.2 Installation

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

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

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

2.1.3 Web Application

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

2.1.4 Programming Examples

The Time Tagger installer provides programming examples for Python, Matlab, 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

Property	Time Tagger 20	Time Tagger Ultra
Termination	50 Ω	50 Ω
Input voltage range	0 to 5 V	-5 to 5 V
Trigger level range	0 to 2.5 V	-2.5 to 2.5V
Minimum signal level	~50 mV	~20 mV
Minimum pulse width	~1 ns	~100 ps

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 40 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.

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

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 photo diodes 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 +/- (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.

More details and explanations can be found in the In Depth Guide More details 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 *TimeTagger 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 40 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.

CHAPTER

FOUR

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 in 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 represent the Time Tagger and one class that represents a base measurement. On top of that, the C++ API also provides all predefined measurements that are made available by the web application and high-level language bindings. To use this API, you have to write and compile a C++ program.

CHAPTER

FIVE

APPLICATION PROGRAMMER'S INTERFACE

5.1 Overview

The API provides methods to control the hardware and to create measurements that are hooked onto the time tag stream. It is written in C++ but wrapper classes for higher level languages (python, Matlab, LabVIEW) are provided, such that the C++ API can directly be used in your application in a way that is equivalent to the C++ classes. The API includes a set of typical measurements that will most likely cover your needs. Implementation of custom measurements can be done in one of two ways:

- Subclassing the Iterator class (best performance, but only available in the C++ API - see example in installation folder)
- Using the TimeTagStream measurement and processing the raw time tag stream.

5.1.1 API documentation

The API documentation in this manual gives a general overview on how to use the *Time Tagger*.

5.1.2 Examples

Often the fastest way to learn how to use an API is by means of examples. Please see the `examples\<language>` subfolder of your *Time Tagger* installation for examples.

5.1.3 Units

Time is measured in ps since device startup and is represented by 64 bit integers. Note that this implies that the time variable will rollover once approximately every 0.83 years since device startup. This will most likely not be relevant to you unless you plan to run your software continuously over one year and are taking data at the instance when the rollover is happening.

5.1.4 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 willobtain 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.

5.1.5 Unused Channels

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

5.2 Organization

The API contains a *small* number of **classes** which you can instantiate in your code. These **classes** are summarized below.

5.2.1 Hardware

TimeTagger Represents the hardware and provides methods to control the trigger levels, input delay, dead time, event filter, and test signals.

5.2.2 Virtual Channels

Combiner Combines two or more channels into one

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

5.2.3 Measurements

Correlation auto- and cross-correlation.

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 an external sampling clock, etc.

Counter Counts 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.

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

HistogramLogBin 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 digitizer

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.

5.3 The TimeTagger class

This class provides access to the hardware and exposes methods to control hardware settings. Behind the scenes it opens the USB connection, initializes the device and receives the time tag stream. Every measurement requires an instance of the TimeTagger class with which it will be associated. In a typical application you will perform the following steps:

1. create an instance of TimeTagger
2. use methods on the instance of TimeTagger to adjust the trigger levels
3. create an instance of a measurement which passes the instance of TimeTagger to the constructor

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 a physical Time Tagger. To implement this in a bullet proof way, TimeTagger instances must be created with a factory function called 'createTimeTagger'. The factory function accepts the serial number of a physical Time Tagger as a string argument (every Time Tagger has a unique hardware serial number). The serial number is the only argument that can be passed. If an empty string or no argument is passed, the first detected Time Tagger will be used. To find out the hardware serial number, you can connect a single Time Tagger, open it and use the 'getSerial' function described below.

The TimeTagger class contains a small number of methods to control the hardware settings, these are summarized below.

5.3.1 Methods

reset() reset the Time Tagger to the startup state

setTriggerLevel(unsigned int channel, double voltage) set the trigger level of an input channel in volts

double getTriggerLevel(unsigned int channel) return the trigger level of an input channel in volts

setInputDelay(unsigned int channel, long long delay) set the input delay of a channel in picoseconds

long long getInputDelay(unsigned int channel) return the input delay of a channel in picoseconds

setFilter(bool state) Deprecated since version V1.0.3: use `setConditionalFilter()` instead.

enable or disable the event filter on the FPGA board. If the filter is active, tags on channel 7 are only transmitted if they were immediately preceded by a tag on channel 0.

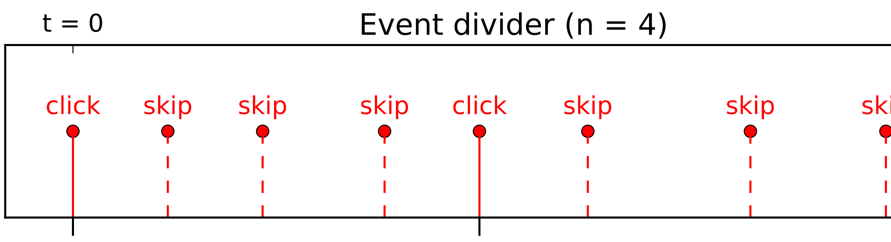
bool getFilter() Deprecated since version V1.0.3: use `getConditionalFilterTrigger()` and `getConditionalFilterFiltered()` instead.

returns true if the event filter on the FPGA board is enabled

setConditionalFilter(<vector int> trigger, <vector int> 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*.

<vector int> getConditionalFilterTrigger() returns the collection of trigger channels for the conditional filter.

<vector int> getConditionalFilterFiltered() returns the collection of channels to which the conditional filter is currently applied.



setEventDivider(int channel, int divider)

applies an event divider filter with the specified factor to a channel. Only every nth event from the original stream passes through the filter, as shown in the image. Note that if a conditional filter is also active, the conditional filter is applied first.

int getEventDivider(int channel) gets the event divider filter factor for the given channel.

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

bool getNormalization() returns true if Gaussian normalization is enabled.

long long setDeadtime(unsigned int channel, long long 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 2.25 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 393.21 µs for the Time Tagger 20 and 147.45375 µs for the Time Tagger Ultra.

long long getDeadtime(unsigned int channel) returns the dead time of a channel in picoseconds

setTestSignal(vector unsigned int, bool state) connect or disconnect the channels with the on-chip uncorrelated signal generator

bool getTestSignal(unsigned int channel) returns true if the internal test signal is activated on the specified channel

string getSerial() returns the hardware serial number

long long getOverflows() returns the number of overflows (missing blocks of time tags due to limited USB data rate) that occurred since startup

long long getOverflowsAndClear() returns the number of overflows that occurred since startup and sets them 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. Synchronize the Time Tagger pipeline, so that all tags arriving after a sync call were actually produced after the sync call. The sync function waits until all historic tags in the pipeline are processed.

5.3.2 Advanced Methods

int getBoardModel() returns the hardware type

int getPcbVersion() returns the hardware version of the device

<vector double> getDACRange() return the minimum and the maximum DAC voltage range

registerChannel(unsigned int channel) enable transmission of time tags on the specified channel

unregisterChannel(unsigned int channel) disable transmission of time tags on the specified channel

unsigned int getChannels() Deprecated since version V1.2.0: use `getChannelList()` instead.

<vector int> getChannelList(int type)

returns the physical channels with the following options

`TT_CHANNEL_RISING_AND_FALLING_EDGES` all channels for the rising and falling edges together

`TT_CHANNEL_RISING_EDGES` the channels of the rising edges

`TT_CHANNEL_FALLING_EDGES` the channels of the rising edges

int getInvertedChannel(int 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.

bool isChannelUnused(int channel) returns true/false in respect to whether the passed channel number is CHANNEL_UNUSED

void setHardwareBufferSize(int size) - TT Ultra only Sets the maximum buffer size within the Time Tagger Ultra. The default value is 64 MTags, but can be changed within the range of 32 kTags to 512 MTags.

autoCalibration(bool verbose=true) run an auto calibration of the Time Tagger hardware using the built-in test signal

2D array long long getDistributionCount() returns the calibration data represented in counts

2D array long long getDistributionPSec() returns the calibration data in picoseconds

long long getPsPerClock() returns the duration of a clock cycle in picoseconds

setTestSignalDivider(int divider) - TT Ultra only change the frequency of the on-chip test signal, the default value 63 correspondss to ~800 kCounts/s

getSensorData() - TT Ultra only prints all avaiable sensor data for the given board

setTimeTaggerChannelNumberScheme(int scheme)

select 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

`TT_CHANNEL_NUMBER_SCHEME_ZERO` force the first channel to be 0

`TT_CHANNEL_NUMBER_SCHEME_ONE` force the first channel to be 1

The method must be applied before the first time tagger instance in created via createTimeTagger()

int getTimeTaggerChannelNumberScheme() returns the used channel schema which is either TT_CHANNEL_NUMBER_SCHEME_ZERO or TT_CHANNEL_NUMBER_SCHEME_ONE

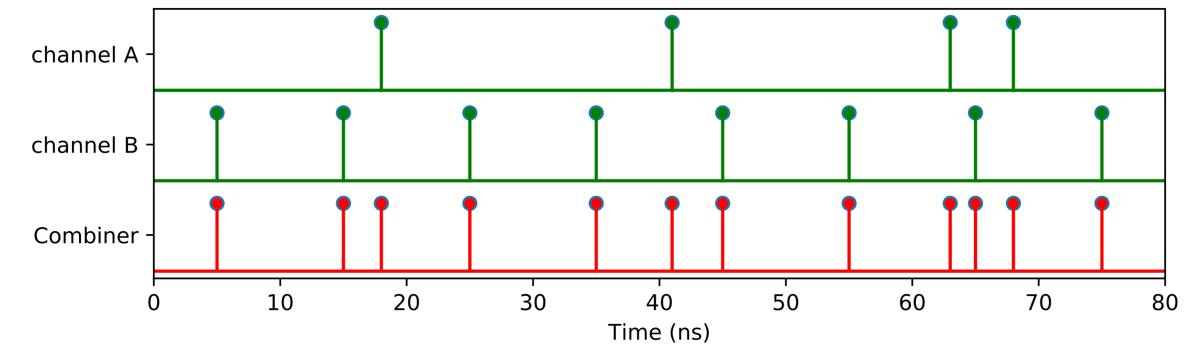
setLED(uint32 bitmask)

manually change the state of the LEDs
illuminate bits:
0 -> LED off
1 -> LED on
0-2: status, rgb - all Time Tagger
3-5: power, rgb - Time Tagger Ultra
6-8: clock, rgb - Time Tagger Ultra
mask bits:
0 -> normal LED behaivor, not overwritten by setLED
1 -> LED state is overwritten by the corresponding bit of 0-8
16-18: status, rgb - all Time Tagger
19-21: power, rgb - Time Tagger Ultra
22-24: clock, rgb - Time Tagger Ultra
The power LED of the Time Tagger 20 cannot be programmed by software.

5.4 Virtual Channels

Virtual channels are software defined as compared to the real input channels, from which they extract data before the modified data is reinjected into the time tag stream. They can be used the same way as real channels by passing the channel number obtained from .getChannel() or .getChannels() from the specific virtual channel object.

5.4.1 Combiner

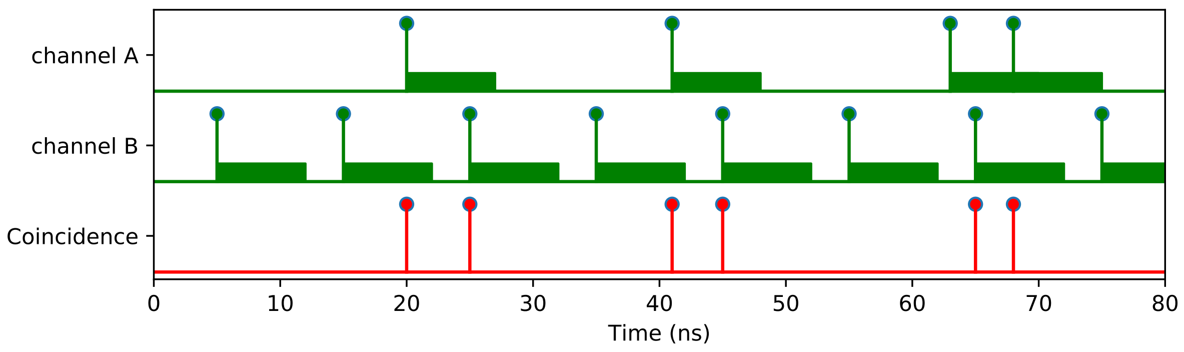


Combines two or more channels into one. The virtual channel is triggered e.g. for two channels when either channel A OR channel B received a signal.

Arguments

channels <vector int> channels which are combined in a single virtual channel

5.4.2 Coincidence



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

Arguments

channels <vector int> channels which are combined in a single virtual channel

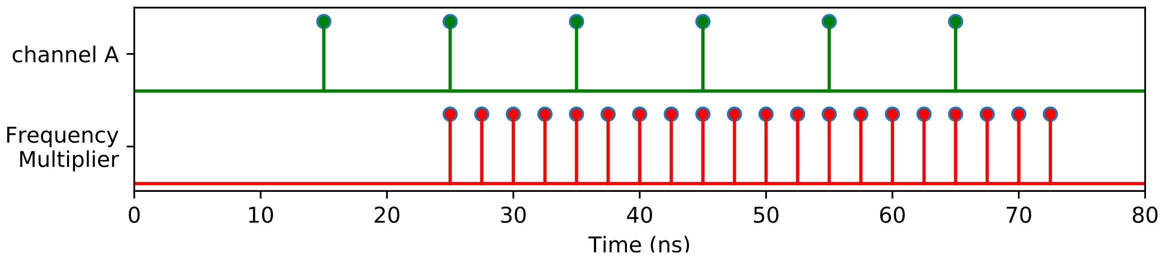
coincidenceWindow max time between all events for a coincidence [ps]

5.4.3 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 a better performance in comparison to multiple virtual Coincidence channels.

Note: Only C++ and python support jagged arrays (uint[][]) which is the underlaying parameter 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), and for LabVIEW a CoincidencesFactory-Class is available to create a Coincidences object, which is also shown in the LabVIEW examples provided with the installer)

5.4.4 FrequencyMultiplier



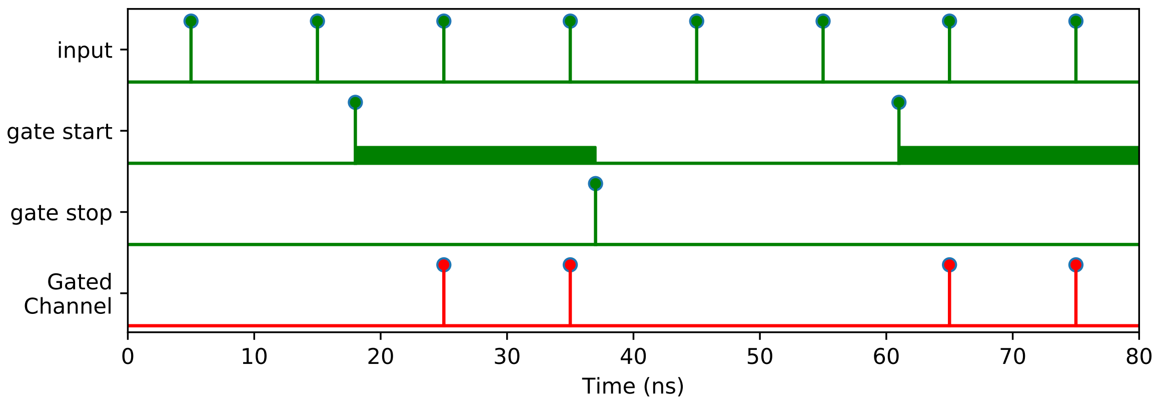
Frequency Multiplier for a channel with a periodic signal

Arguments

input_channel channel on which the upscaling of the frequency is based on

multiplier frequency upscaling factor

5.4.5 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.

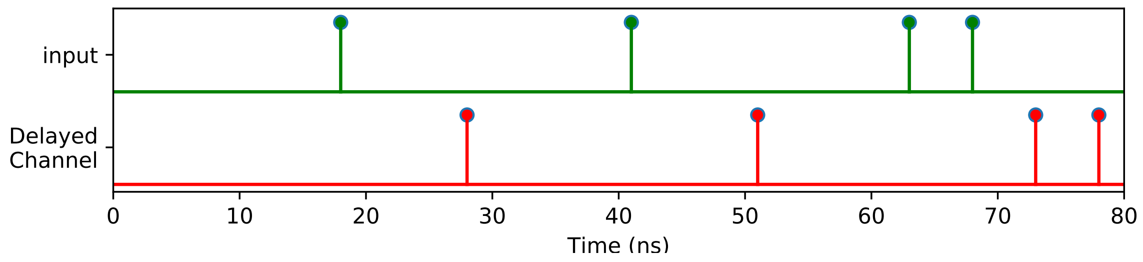
Arguments

input_channel channel which is gated

gate_start_channel channel on which a signal detected will start the transmission of the input_channel through the gate

gate_stop_channel channel on which a signal detected will stop the transmission of the input_channel through the gate

5.4.6 DelayedChannel



Clones an input channel, which can be delayed via the delay parameter in the constructor or the setDelay(delay) method. Note that internally buffered data might get lost when setDelay() is called with a reduced delay time.

Note: If a global delay of the channel is required set, it is recommended to use tagger.setInputDelay(channel, delay) instead of creating a virtual DelayedChannel. The method .setInputDelay(…) requires less CPU performance compared to DelayedChannel.

Arguments

input_channel channel which is delayed

delay amount of time to delay in ps, must be positive

5.5 Measurement Classes

The library includes a number of common measurements that will be described in this section. 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. The methods are summarized below.

5.5.1 Methods common to all Measurements

getData() Returns the data accumulated up until now. The returned data can be a scalar, vector or array, depending on the measurement.

clear() reset the accumulated data to an array filled with zeros

start() start data acquisition

startFor(duration, clear = True) starts/continues the data acquisition for the given duration (in ps). After the duration, 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'.

stop() stop data acquisition

getCaptureDuration() total capture duration since the last call to clear

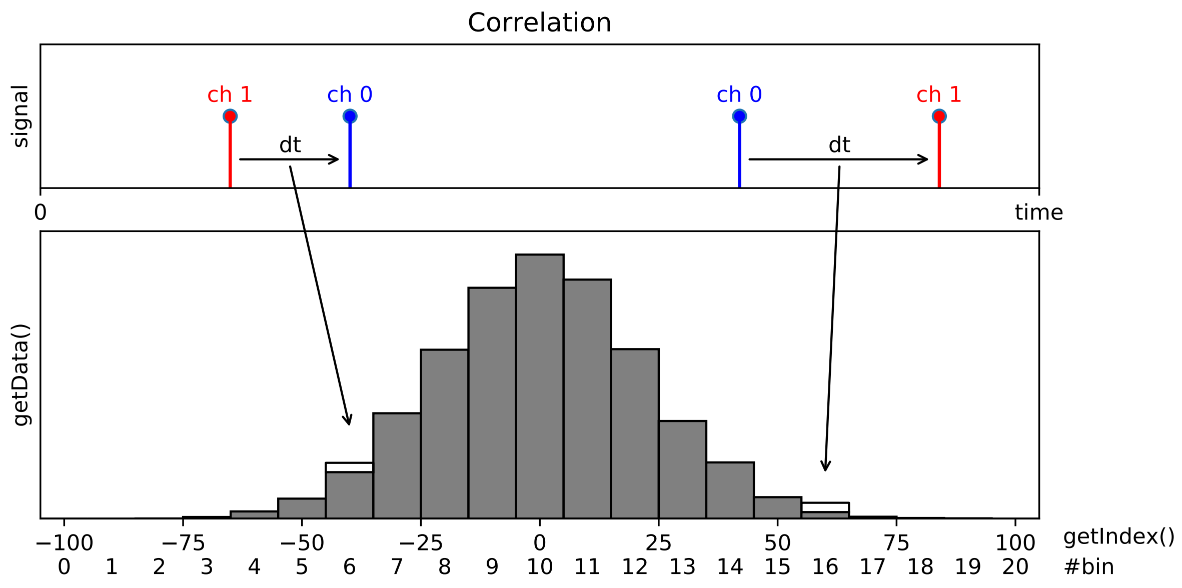
Attention: All measurements start accumulating data immediately after their creation.

In a typical application, the following steps are performed:

1. create an instance of a measurement, e.g.~a count rate on channel 0
2. wait for some time
3. retrieve the data accumulated by the measurement up until now by calling the 'getData' method.

The specific measurements are described below.

5.5.2 Correlation



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

Arguments

tagger <reference> reference to a Time Tagger

channel 1 <int> reference channel

channel 2 <int> 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 <longlong> bin width in ps

n_bins <int> the number of bins in the resulting histogram

Methods

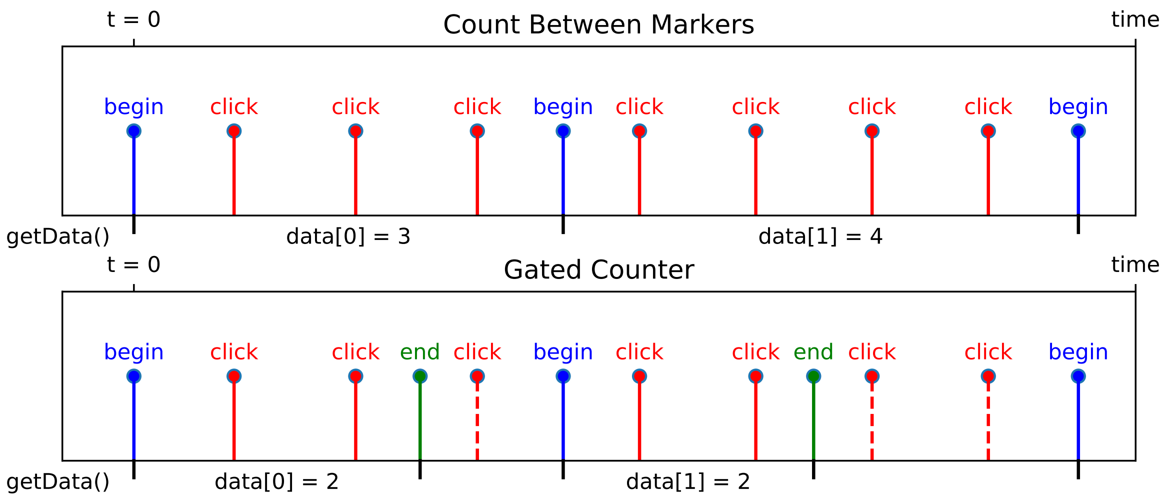
getData() returns a one-dimensional array of size n_bins containing the histogram

getDataNormalized() (V1.0.20, beta) return the normalized data according to this publication

getIndex() returns a vector of size 'n_bins' containing the time bins in ps

clear() resets the accumulated date

5.5.3 CountBetweenMarkers



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

Arguments

tagger <reference> reference to a Time Tagger

click_channel <int> channel on which clicks are received, gated by begin_channel and end_channel

begin_channel <int> channel that triggers beginning of counting and stepping to the next value

end_channel <int> channel that triggers end of counting

n_values <int> number of values stored (data buffer size)

Methods

getData() returns an array of size ‘n_values’ containing the acquired counter values

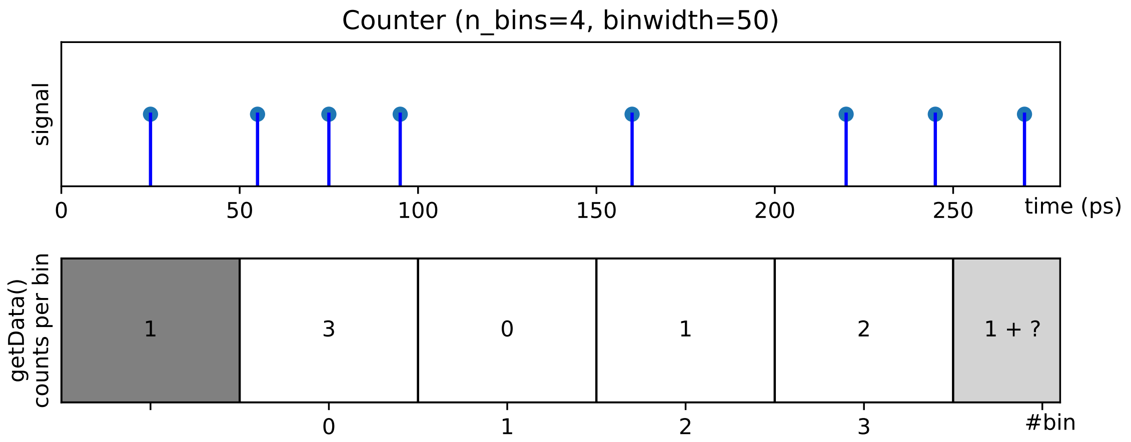
getIndex() returns a vector of size ‘n_values’ containing the time in ps of each start click in respect to the very first start click

getBinWidths() returns a vector of size ‘n_values’ containing the time differences of ‘start -> (next start or stop)’ for the acquired counter values

clear() resets the array to zero and restarts the measurement

ready() returns ‘true’ when the entire array is filled

5.5.4 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.

Arguments

tagger <reference> reference to a Time Tagger

channels <vector int> channels used for counting tags

binwidth <longlong> bin width in ps

n_values <int> number of bins (data buffer size)

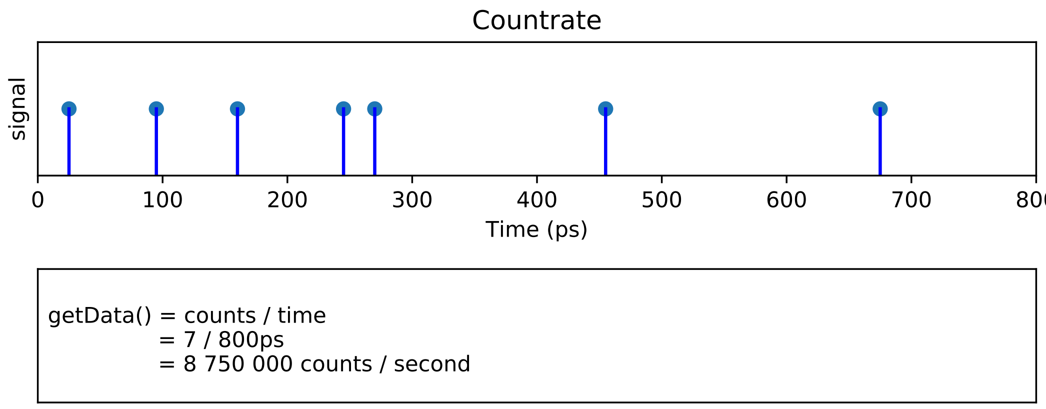
Methods

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

getIndex() returns a vector of size 'n_values' containing the time bins in ps

clear() resets the array to zero and restarts the measurement

5.5.5 Countrate



```
getData() = counts / time
          = 7 / 800ps
          = 8 750 000 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 of the measurement.

Arguments

tagger <reference> reference to a Time Tagger

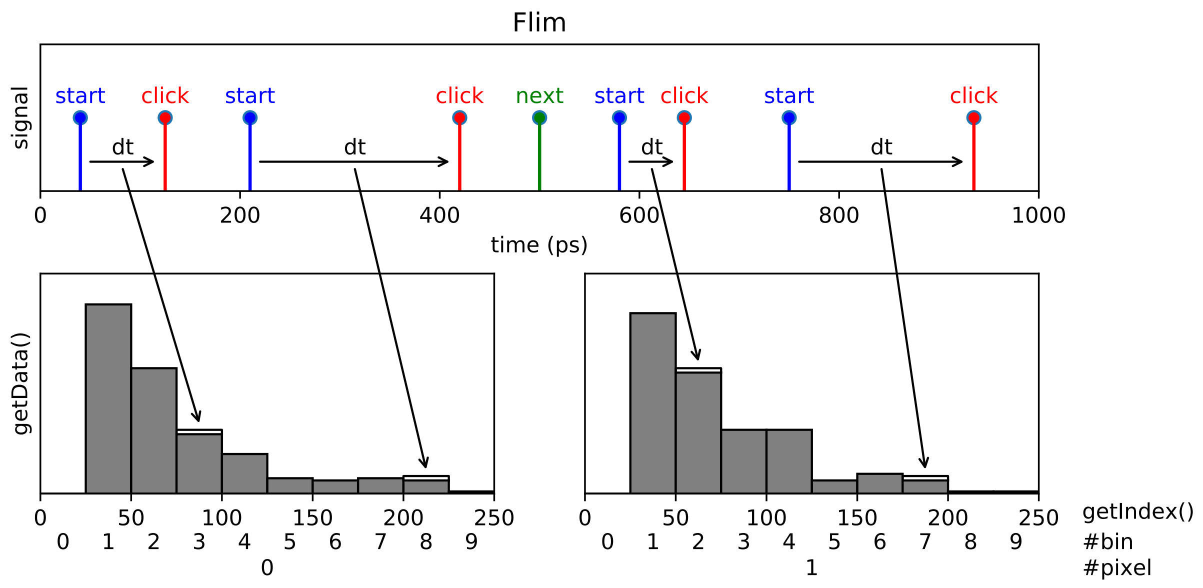
channels <vector int> channels for which the average count rate is measured

Methods

getData() returns the average count rate in counts per second

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

5.5.6 FLIM



Fluorescence-lifetime imaging microscopy or 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 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.

Arguments

tagger <reference> reference to a Time Tagger

click channel <int> channel on which clicks are received

start channel <int> channel on which start clicks are received

next_channel <int> channel on which pixel triggers are received

binwidth <longlong> bin width in ps

n_bins <int> number of bins in each histogram

n_pixels <int> number of pixels

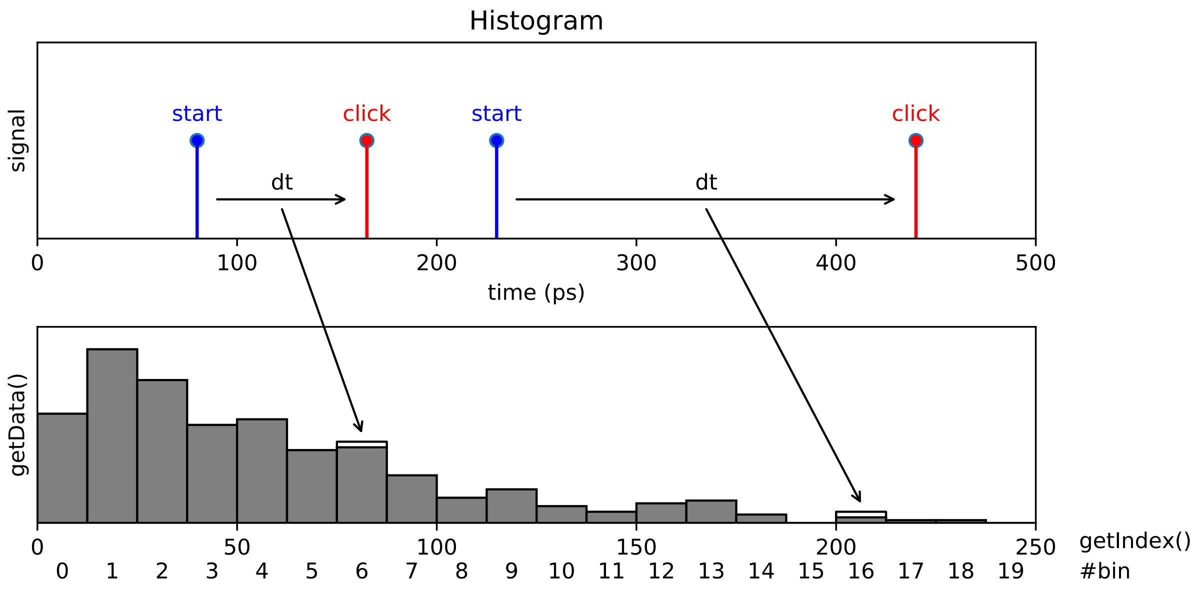
Methods

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

getIndex() returns a vector of size ‘n_bins’ containing the time bins in ps.

clear() resets the array to zero.

5.5.7 Histogram



Accumulate time differences into a histogram. This is a simple multiple start, multiple stop measurement. This is a special case of the more general ‘TimeDifferences’ measurement. Specifically, the measurement waits for clicks on the ‘start channel’, and for each start click, it measures the time difference between the start click and all subsequent clicks on the ‘click channel’ and stores them in a histogram. The histogram range and resolution is 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.

Arguments

tagger <reference> reference to a Time Tagger

click channel <int> channel on which clicks are received

start channel <int> channel on which start clicks are received

binwidth <longlong> bin width in ps

n_bins <int> the number of bins in the histogram

Methods

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

getIndex() returns a vector of size ‘n_bins’ containing the time bins in ps.

clear() resets the array to zero.

5.5.8 HistogramLogBins

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

Arguments

tagger <reference> reference to a Time Tagger

click channel <int> channel on which clicks are received

start channel <int> channel on which start clicks are received

exp_start <longlong> exponent 10^exp_start in seconds where the very first bin begins

exp_stop <longlong> exponent 10^exp_stop in seconds where the very last bin ends

n_bins <int> the number of bins in the histogram

Methods

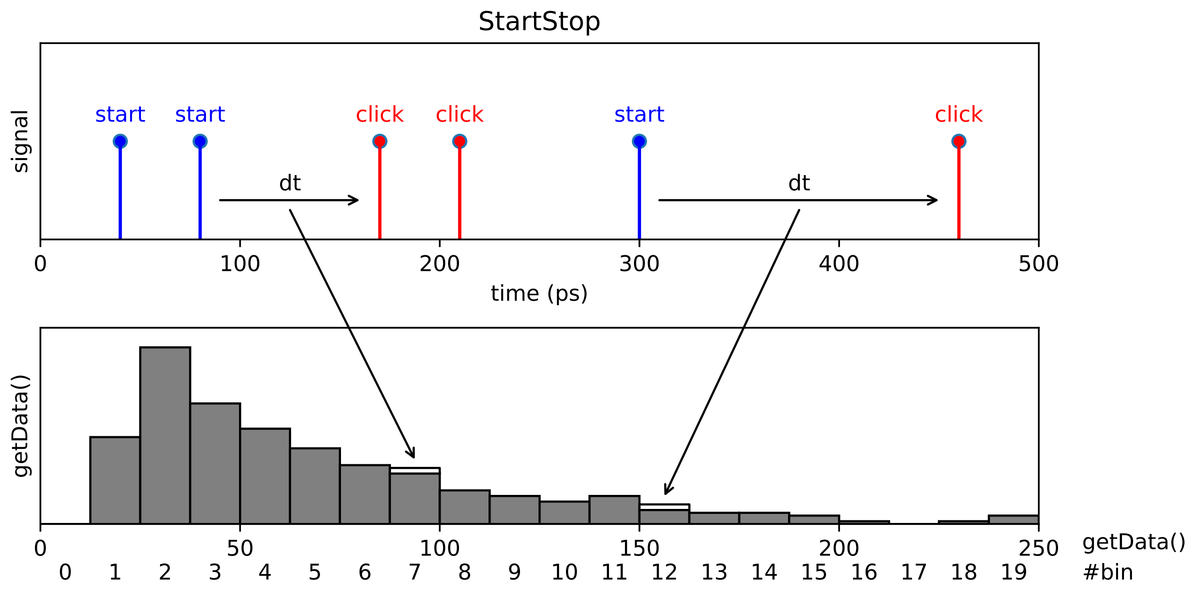
getData() returns a one-dimensional array of size n_bins containing the histogram

getDataNormalzed() returns the counts normalized by the binwidth of each bin

getBinEdges() returns a vector of size 'n_bins+1' containing the bin edges

clear() resets the array to zero

5.5.9 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 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.

Arguments

tagger <reference> reference to a Time Tagger

click_channel <int> channel on which stop clicks are received

start_channel <int> channel on which start clicks are received

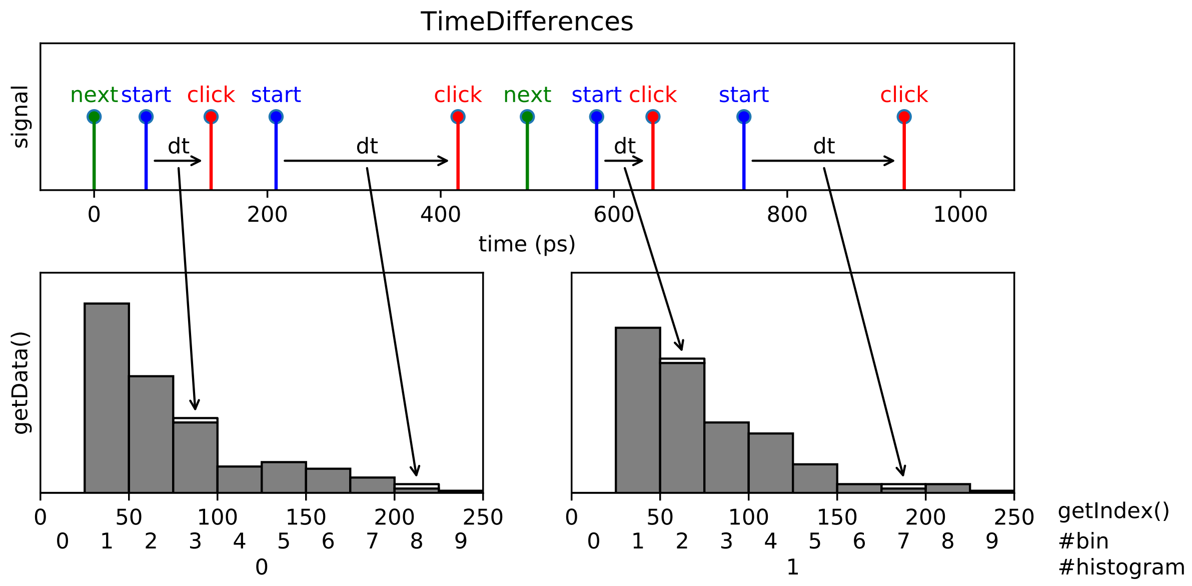
binwidth <longlong> bin width in ps

Methods

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

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

5.5.10 TimeDifferences



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

The data obtained from subsequent start tags can be accumulated into the same histogram (one-dimensional measurement) or into different histograms (two-dimensional measurement). In this way, you can perform more general two-dimensional time-difference measurements. The parameter 'n_histograms' specifies the number of histograms.

After each tag on the 'next_channel', the histogram index is incremented by one and reset to zero after reaching the last valid index. The measurement starts with the first tag on the 'next_channel'.

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

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

Arguments

tagger <reference> reference to a Time Tagger

click channel <int> channel that increments the count in a bin

start channel <int> channel that sets start times relative to which clicks on the click channel are measured

next channel <int> channel that increments the histogram index

sync channel <int> channel that resets the histogram index to zero

binwidth <longlong> bin width in ps

n_bins <int> number of bins in each histogram

n_histograms <int> number of histograms

Methods

getData() returns a two-dimensional array of size 'n_bins' by 'n_histograms' containing the histograms

getIndex() returns a vector of size 'n_bins' containing the time bins in ps

clear() resets all data to zero

setMaxCounts() set the number of rollovers at which the measurement stops integrating

getCounts() returns the number of rollovers (histogram index resets)

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

Overflow handling

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

sync_channel and next_channel are 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 undefined the accumulation continues

5.5.11 Dump

Dump the time tag stream to a file in a binary format.

Each data block transferred to the PC is stored in a file. The data blocks are stored in the following data format:

- 8 bit unsigned: overflow (0 == false or 1 == true)
- 24 bit reserved
- 32 bit unsigned: channel number
- 64 bit: time in ps from device startup

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

Arguments

tagger <reference> reference to a Time Tagger

filename <str> name of the file to dump to

max_tags <int> stop after this number of tags has been dumped

channels <vector int> channels which are dumped to the file (when empty or not passed all active channels are dumped)

5.5.12 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.

Arguments

tagger <reference> reference to a Time Tagger

max_tags <int> buffer size for storing time tags

channels <vector int> channels which are dumped to the file (when empty or not passed all active channels are dumped)

Methods

getData() <TimeTagStreamBuffer> returns a TimeTagStreamBuffer object which contains a vector for the overflows, channels and the timestamps in ps of the tags. The hasOverflows field shows whether an overflow was detected in any of the tags received.

Example (Python)

```python
from TimeTagger import createTimeTagger, TimeTagStream, TimeTagStreamBuffer
from time import sleep
tagger = createTimeTagger()
# enable test signal
tagger.setTestSignal([0, 1], True)
tagger.sync()
```

(continues on next page)

(continued from previous page)

```
# the received tags will be copied to the buffer object
buffer = TimeTagStreamBuffer()
# capture the incoming tags (with a maximum of 1M tags)
stream = TimeTagStream(tagger, 1000000, [0, 1])
# take data for 0.5s
sleep(0.5)
stream.getData(buffer)
print ("Total number of tags stored in the buffer: " + str(buffer.size));
print ("Show the first 10 tags");
for i in range(min(buffer.size, 10)):
    print("  time in ps: " + str(buffer.tagTimestamps[i]) + " signal received on␣
↪channel: "  + str(buffer.tagChannels[i]))
```

CHAPTER
SIX

IN DEPTH GUIDES

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.

```
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.

```
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 40 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,

```
Correlation(timeTagger, 0, 7, ...)
```

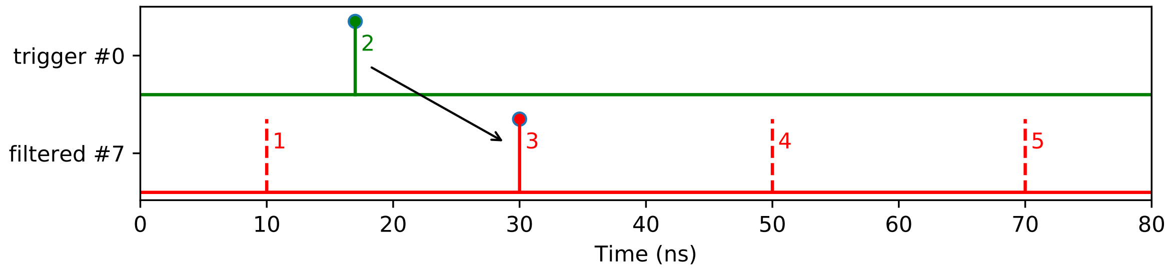
the Time Tagger world ran into data overflows without the filter applied. Overflows are indicated by a red LED or can be retrieved by calling `tagger.getOverflows()`. The correlation measurement would still work but the effective correlation rate drops by a factor of (total input rate / max transfer 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 >= (deadtime + 3 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.

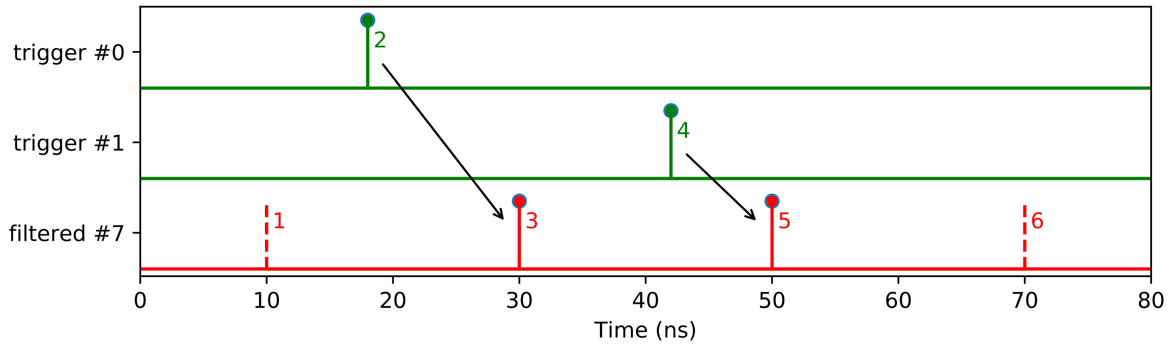
To have event 2 and 3 on top of each other for the evaluation (e.g. correlation measurement), `setInputDelay(0, dt)` can be applied.

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])
```

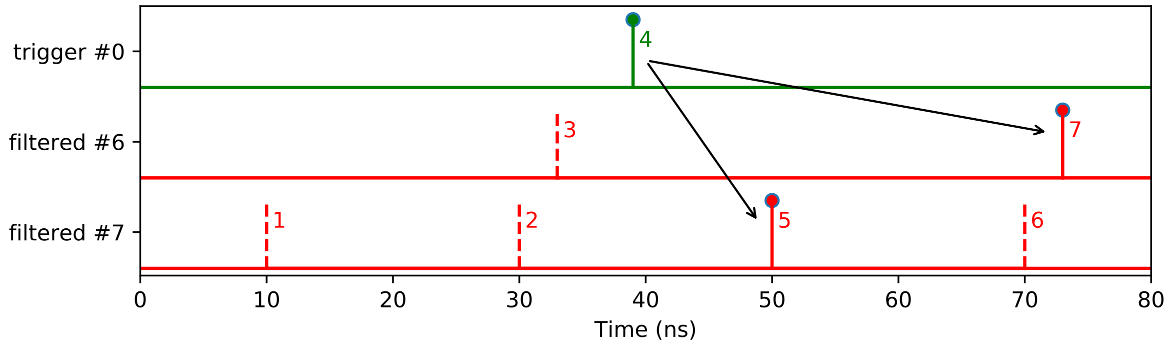


Because of event 2 event 3 is transmitted, and because of event 4 event 5 is transmitted.

6.2.2 Multiple filtered channels

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

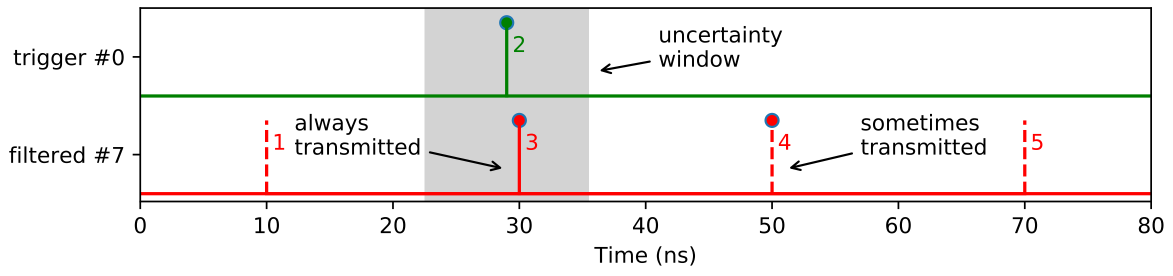
```
setConditionalFilter(trigger=[0], filtered=[6,7])
```



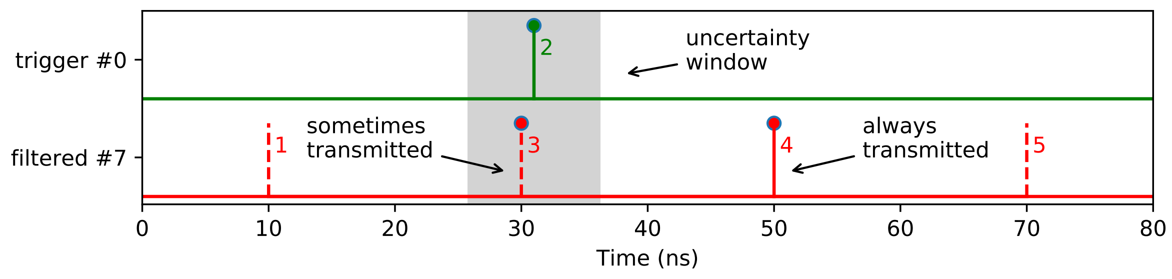
All the very next events of the filtered-channels will be transmitted (event 5 of #7 and event 7 of #6) after an event at the trigger-channel (event 4).

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:

```
setConditionalFilter([], [])
```

CHAPTER

SEVEN

LINUX

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

Graphical user interface (web application):

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

Known issues

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

Using the Time Tagger with Python 2.7, 3.5 or 3.6:

- 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.

General remark:

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

CHAPTER

EIGHT

REVISION HISTORY

8.1 V2.2.4 - 29.01.2019

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

8.2 V2.2.2 - 13.11.2018

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

8.3 V2.2.0 - 07.11.2018

General improvements

- support for devices starting with channel 1 instead of 0
- under certain cirmunstances, 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 to analyze incoming tags with logarithmic bin sizes.
- ‘FrequencyMultiplier’ virtual channel class for upscaling a signal attached to the Time Tagger. This method can be used as an alternative to the ‘Conditonal Filter’.

- ‘SynchronizedMeasurments’ 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

8.4 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

8.5 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

8.6 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

8.7 V2.1.0 - 06.03.2018

Time Tagger Ultra

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

8.8 V2.0.4 - 01.02.2018

Bug fixes

- Closing the web application server window works properly now

8.9 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.

8.10 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

8.11 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

8.12 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 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 trough 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

8.13 V1.0.4 - 24.11.2016

Hardware changes

- extended event filter to multiple condition and filter channels
- improved jitter for channel 0
- channel delays might be different from 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

8.14 V1.0.2 - 28.07.2016

Major changes:

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

API changes:

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

8.15 V1.0.0

initial release supporting python

8.16 Channel Number Schema 0 and 1

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

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

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

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

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

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

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