Time Tagger Documentation

Release 1.0

Swabian Instruments

June 29, 2015

CONTENTS

1 Installation instructions **3**
1.1 Windows users . . . 3

2 Getting Started **5**
2.1 Windows users . . . 5

3 Hardware **7**
3.1 Input channels . . . 7
3.2 Data connection . . . 7
3.3 Status LEDs . . . 7
3.4 Test signal . . . 8
3.5 Virtual channels . . . 8
3.6 Synthetic input delay . . . 8
3.7 Synthetic dead time . . . 8
3.8 Event filter . . . 8
3.9 Bin equilibration . . . 8
3.10 Overflows . . . 9
3.11 General purpose IO (available upon request) . . . 9

4 Software Overview **11**
4.1 Web application and JSON-RPC interface . . . 11
4.2 Precompiled libraries and high level language bindings . . . 11
4.3 C++ API . . . 11

5 Application Programmer's Interface **13**
5.1 Overview . . . 13
5.2 Organization . . . 13
5.3 The TimeTagger class . . . 14
5.4 Measurement Classes . . . 15
5.5 Defining custom measurements by subclassing Iterator . . . 19

6 Indices and tables **21**

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

1.1 Windows users

1.1.1 Requirements

Operating System

All 32 and 64 bit windows versions since windows XP are supported.

Python

If you want to run the **quickstart guide**, or the **web application** (currently under development), you need the following additional third-party software packages (available for free).

- **python 2.7.x** or **python 3.4.x** installed along with **ipython**
- the following python modules: **numpy, matplotlib (pylab), time, cPickle**

On windows it is easiest to install a **python distribution** that contains all required packages by default. The **anaconda** or **canopy** python distributions will cover this. If you don't have python installed, please download and install now either of these python distributions. Note that **anaconda** is available as **python 3.4.x**.

Note: If you are using a fresh install of the **canopy** distribution, please make sure to run the graphical interface once from the start menu to have your python paths set up corectly.

Installation

Note: If you intend to use the *TimeTagger 20* with the Python programming language, please make sure that you install python first as described above. This ensures that the installer detects your Python installation automatically.

1. Download and run the most recent *TimeTagger 20* windows installer.

Connect the *TimeTagger 20* to your computer with the USB cable.

Caution: Wait until windows has recognized the device and finished installing the driver.

You should now be ready to use your *TimeTagger 20*.

CHAPTER

TWO

GETTING STARTED

2.1 Windows users

1. Make sure the *TimeTagger 20* software and a Python distribution are installed and the *TimeTagger 20* connected to your computer (see previous section).
2. Open a command shell and `cd` to the `..\examples\python\quickstart` folder under your *TimeTagger 20* installation directory
3. Start an **ipython** shell with plotting support by entering `ipython --pylab`
4. Run the **quickstart.py** script by entering `run quickstart`

The script demonstrates a selection of the features provided by the *TimeTagger 20* programming interface and runs some example measurements using the built in test signal generator and plots the result.

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

Among others, the script will...

1. Create an instance called 'tagger' that represents the device.
2. Start the builtin test signal (~0.8 MHz square wave) and apply it to channels 0 a nd 1
3. Create a time trace of the click rate on the first two channels, 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. Show you how to create virtual channels, use synchronization, event filter and control the input trigger level.

2.1.1 Where to go from here

To learn more about the *TimeTagger 20* you are encouraged to consult the following resources.

1. If you have not done so already, have a look at the Python script you just run.
2. More details about the software interface are covered by the C++ API documentation in the subsequent section
3. You are also encouraged to study the C++ source code provided in the *TimeTagger 20* install directory (released under the GPL)
4. Code examples in `..\examples\python\traits` show you how to quickly implement an interactive graphical interface in python

Note: The python GUI examples, additionally require the following python packages installed on your system: *traits, traitsui, chaco, pyface enable*.

CHAPTER

THREE

HARDWARE

3.1 Input channels

The *TimeTagger 20* has 8 SMA connectorized input channels numbered 0 to 7 throughout this document. The electrical characteristics are tabulated below. Both rising and falling edges are detected on the input channels. On the software level, rising edges correspond to channel numbers 0 to 7 and falling edges correspond to respective channel numbers 8 to 15. Thereby, you can treat rising and falling edges in a fully equivalent fashion.

3.1.1 Electrical characteristics

Property	Value
Termination	50 Ω
Input voltage range	0 to 5 V
Trigger level range	0 to 3.3 V
Minimum signal level	~50 mV
Minimum pulse width	~1 ns

3.2 Data connection

A USB connection is used for data and power supply. Please ensure that the USB port is capable of providing the full specified current (500 mA). A USB 2.0 data connection is required for reasonable performance. Operating the device via a USB hub is strongly discouraged. The *TimeTagger 20* can stream about 5 M tags per second.

3.3 Status LEDs

The *TimeTagger 20* 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 *TimeTagger 20* has a built in test signal generator that generates a square wave with a frequency in the range 0.9 to 1.0 MHz. You can apply the test signal to any input channel instead of the external input. This is useful for testing, calibrating and setting up,

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

3.7 Synthetic dead time

You can introduce for each channel a synthetic dead time. 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 Event filter

In a typical fluorescence lifetime application, a target is stimulated with laser pulses with a fast repetition rate, typically in the range 10 - 100 MHz. Electrical synchronization pulses are generated that are simultaneous with the excitation laser pulses and are sent to the *TimeTagger 20* on one channel, while single photon clicks emitted from the target are sent to another channel. Because the data rate of the synchronization pulses is so high, streaming and processing all generated time tags by the computer is not possible - and not necessary, since only those synchronization pulses are of interest that are followed by a photon event. It is therefore desirable to discard all synchronization time tags in the data stream except those that are followed by a photon. Since the synchronization pulses are periodic with a very well defined period, it is equivalent to keep only those synchronization time tags that are {em preceded} by a photon.

This feature is implemented by an event filter that is currently hardcoded between channel 0 and channel 7. It is assumed that photon clicks are entering channel 0 and laser sync clicks are entering channel 7. When the filter is active, time tags on channel 7 are only passed if a time tag has been registered on channel 0 before. Subsequent tags are discarded until the next tag on channel 0 is detected.

This filter is all you need to perform fluorescence lifetime measurements and fluorescence lifetime imaging. If you are interested in event filters with more complex logic, please contact us for custom designs.

3.9 Bin equilibration

Discretization of electrical signals is never perfect. In time-to-digital conversion, this is manifest as small differences (few ps) of the bin sizes inside the converter that even varies from chip to chip. This imperfection is inheret to any time-to-digital conversion hardware. It is usually not apperent to the user. However, when correlations between two channels are measured on short time scales you might see this as a weak periodic ripple ontop of your signal. If you wish to turn off this ripple, you can enable an equilibration of the bin sizes at the cost of a decrease of the time resolution by $\sqrt{2}$. This feature is disabled by default.

3.10 Overflows

Data rates are never infinite. The *TimeTagger 20* is capable of streaming about 5 M tags per second, At higher rates, data loss occurs and parts of the time tags are lost. The hardware allows you to check whether an overflow condition has occured. If no overflow occurrs, you can be sure that every time tag is received.

3.11 General purpose IO (available upon request)

The device is ready to be equipped with up to four SMA connectorized general purpose IO ports 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, etc.. Please contact us for custom designs.

CHAPTER

FOUR

SOFTWARE OVERVIEW

The heart of the *TimeTagger 20* 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. E.g., 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, etc.. This is a powerful architecture that allows you to perform any thinkable digital time domain measurement in real time. You have several choices to use this architecture.

4.1 Web application and JSON-RPC interface

Note: This feature is currently under development. Please contact us if you would like to use the timetagger in this way.

The easiest way of using the *TimeTagger 20* 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 and get life plots, save and load the acquired data from within a web browser. In addition, you can also access and reomte control the web application via a JSON-RPC interface.

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 TimeTagger even easier, we have equipped these libraries with bindings to higher level languages, specifically Python so that you can directly use the TimeTagger from this language. 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 code in the high level language of your choice. Refer to chapter *quickstart* and *Application Programmer's Interface* if you plan to use the TimeTagger in this way.

Note: Bindings for other languages such as Java, Matlab and Labview are available upon request.

4.3 C++ API

The underlying software architecture is provided by a C++ API that implements two classes: one class that represent the TimeTagger and one class that represents a base measurement. Ontop 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. If you want to implement a custom measurement you need to follow this approach. Refer to chapter *Application Programmer's Interface* if you plan to use the TimeTagger in this way.

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, specifically python can be 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 is based on subclassing from a C++ base class and thus is only available in the C++ API.

5.1.1 API documentation

The API documentation in this manual gives a general overview how to use the *TimeTagger 20*. More detailed information can be found in the API reference (generated with Doxygen). .. ToDo reference to doxygen stuff

5.1.2 Examples

Often the fastest way to learn how to use an API is by means of examples. Please see the `\exsamples` subfolder of your *TimeTagger 20* installation for examples.

5.1.3 Units

Time is measured in ps since device startup and represented by 64 bit integers. Note that this implies that the time variable will rollover once after about 0.83 years. 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 *TimeTagger 20* to detect both rising and falling edges. Throughout the software API, the rising edges are represented by channels 0 to 7 and the falling edges are represented by channel numbers 8 to 15. Virtual channels will obtain numbers from 16 onwards.

5.2 Organization

The API contains a *small* number of **classes** which you 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 channels into one

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

5.2.3 Measurements

Iterator Base class for implementing custom measurements.

Countrate Average tag rate on one or more channels.

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

CountBetweenMarkers Counts tags on one channel where the bins are determined by triggers on one or two other channels. Uses static buffer output. Use this to implement a gated counter, a counter synchronized to an external sampling clock, etc.

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

Histogram A simple histogram of time differences. This can be used e.g. to measure lifetime.

Correlation auto- and cross-correlation.

FLIM Fluorescence lifetime imaging.

StartStop Accumulates a histogram of time difference between pairs of tagss on two channels. Only the first stop tag after a start tag is considered. Subsequent stop tags are discarded. The Histogram length is unlimited.

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 to 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 passing the instance of TimeTagger to the constructor

You can use multiple TimeTaggers on one computer simultaneously. In this case, you usually want to associate your instance of the TimeTagger class to a physical TimeTagger. 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 TimeTagger as a string argument (every TimeTagger has a unique hardware serial number). The serial number is the only argument that can be passed. If an mpty string or no argument is passed, the first detected TimeTagger will be used. To find out the hardware serial number, you can connect a single timetagger, open it and use the ‘getSerial’ function described below.

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

5.3.1 Methods

setTriggerLevel() set the trigger level of an input channel

getTriggerLevel() return the trigger level of an input channel

setInputDelay() set the input delay of a channel

getInputDelay() return the input delay of a channel

setFilter() enable or disable the laser synchronization filter (currently hardcoded between channel 0 and 7)

getFilter() return the state of the laser synchronization filter

setNormalization() activate or deactivate gaussian normalization of the detection jitter

getNormalization() return whether input normalization is turned on

setDeadTime() set the dead time of a channel

getDeadTime() return the dead time of a channel

setTestSignal apply internal test signal to a channel

getTestSignal check whether the internal test signal is applied to a channel

getSerial() return the hardware serial number

getOverflows() return the number of overflows that occurred since startup

5.4 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. You can use this base class to write custom measurments in C++, as described in *subclassing*.

All measurements provide a small number of methods to start and stop the excecution and to access the accumulated data. The methods are summarized below.

5.4.1 Methods common to all Measurements

getData() Returns the data accumulated up to 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

stop() stop data acquisition

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

In a typical application you will perform the following steps:

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

The specific measurements are described below.

5.4.2 Countrate

Measures the average countrate on one or more channels. Specifically, it counts tags on the specified channels and determines the time between the first tag since instantiation and the latest tag. The ratio of the number of tags and the time corresponds to the average countrate since the first tag.

Arguments

tagger <reference> reference to a time tagger

channels <vector int> channels used for counting tags

Methods

getData() returns the average countrate in counts per second.

clear() resets the accumulated counts to zero and uses the next incoming tag as the first tag.

5.4.3 Counter

Time trace of the countrate 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' times 'n_values'. The array is treated as a circular buffer that is, all values in the array are shifted by on 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> binwidth in ps

n_values <int> number of values

Methods

getData() returns an array of size 'number of channels' times '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.

5.4.4 CountBetweenMarkers

Countrate 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 measurement stops when all entries in the data vector are filled.

Arguments

tagger <reference> reference to a time tagger

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

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 bins in ps.

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

ready() returns 'true' when the entire array is filled.

5.4.5 TimeDifferences

A multidimensional histogram measurement optionally with up to three additional channels that control how to step through the indices of the histogram array. This is a very powerful and generic measurment. 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 of 'n_bins' bins of binwidth '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 refered to as 'single 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. You can also provide a synchronization trigger that resets the histogram index by specifying a 'sync_channel'.

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 both for a one-dimensional and for a two-dimensional measurement, it will measure until every histogram has a seen the specified number of start 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 s 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> binwidth 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’ times ‘n_histograms’ containing the histograms.

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

clear() resets the array to zero.

setMaxCounts() set the maximum number of start clicks accepted

getCounts() returns the number of start clicks

ready() returns ‘true’ when the required number of start clicks set by ‘setMaxCounts’ has been reached

5.4.6 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> first channel

channel 2 <int> second channel

binwidth <longlong> binwidth in ps

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

Methods

getData() returns a one-dimensional array of size 2*n_bins+1 containing the histograms.

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

clear() resets the array to zero.

setMaxCounts() set the maximum number of start tags accepted

getCounts() returns the number of start tags

ready() returns ‘true’ when the required number of start tags set by ‘setMaxCounts’ has been reached

5.4.7 FLIM

Fluorescence lifetime imaging. Specifically, the measurement performs a single-start-multiple-stop measurement and accumulates the time differences into a histogram with specified binwidth and number of bins. The condition for moving to the next pixel can either be a pixel trigger on a third channel or a predefined accumulation time per pixel. After accumulating a number of 'pixels' histograms, the measurement stops. This measurement is also useful to record cross-correlation on multiple pixels.

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

binwidth <longlong> binwidth 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 'pixels' times 'bins' containing the histograms.

getIndex() returns a vector of size 'n_pixels' containing the pixel times in ps.

clear() resets the array to zero.

ready() returns 'true' when the measurement is ready

5.4.8 Dump

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

Arguments

<str> filename name of the file to dump to

Methods

stop() stop the measurement

5.5 Defining custom measurements by subclassing Iterator

This section is under construction..

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*