



Time Tagger User Manual

Release 2.17.4.0

Swabian Instruments

Jul 16, 2024

CONTENTS

1	Getting Started	1
1.1	Get familiar with your Time Tagger	1
1.2	Graphical User Interfaces	2
1.2.1	Time Tagger Lab	2
1.2.2	Web Application	4
1.3	Programming languages	4
1.3.1	Python	4
1.3.2	LabVIEW (via .NET)	7
1.3.3	Matlab (wrapper for .NET)	7
1.3.4	Wolfram Mathematica (via .NET)	7
1.3.5	.NET	7
1.3.6	C#	8
1.3.7	C++	8
2	Installation instructions	11
2.1	Windows	11
2.1.1	Installation	11
2.2	Linux	11
2.2.1	Installation	11
2.2.2	Known issues	12
2.2.3	Time Tagger with Python	12
2.2.4	Time Tagger with C++	12
2.3	Hardware license upgrades	12
2.3.1	Time Tagger Lab	13
2.3.2	Web Application	14
3	Tutorials	17
3.1	Confocal Fluorescence Microscope	17
3.1.1	Time Tagger configuration	18
3.1.2	Intensity scanning microscope	19
3.1.3	Fluorescence Lifetime Microscope	20
3.1.4	Alternative pixel trigger formats	21
3.2	Optically Detected Magnetic Resonance	24
3.2.1	Creation of optical and microwave pulse patterns	25
3.2.2	Signal generation and detection	27
3.2.3	Sweeping modes	28
3.2.4	ODMR contrast	29
3.3	Measuring Coincidences	30
3.3.1	Time Tagger configuration	31
3.3.2	Coincidence-counting	32

3.3.3	Delay adjustment for coincidence detection	33
3.3.4	Exclusive coincidences using Combinations virtual channel	34
3.3.5	Coincidence-counting vs Correlation Measurement	36
3.4	Remote Time Tagger with Python	37
3.4.1	Sharing a Time Tagger with Network Time Tagger	37
3.4.2	Remote control of a Time Tagger with Pyro	38
3.4.3	Remote procedure call	39
3.4.4	Initial setup	39
3.4.5	Minimal example	39
3.4.6	Creating the Time Tagger	40
3.4.7	Measurements and virtual channels	42
3.4.8	Working example	43
3.4.9	What is next?	45
4	Synchronizer	47
4.1	Overview	47
4.2	Key applications	47
4.2.1	Crosstalk elimination	47
4.2.2	High transfer rate	47
4.2.3	Multi-room experiment	48
4.2.4	Synchronizer with a single Time Tagger	48
4.3	Requirements	48
4.4	Cable connections	48
4.4.1	Using an external reference clock	50
4.5	Software and channel numbering	50
4.5.1	Incomplete cable connections	51
4.5.2	Buffer overflows	51
4.6	Limitations	51
4.6.1	Conditional filter	51
4.6.2	Internal test signal	51
4.7	Status LEDs and troubleshooting	52
5	Hardware	53
5.1	Input channels	53
5.1.1	Electrical characteristics	53
5.1.2	Configurable input termination - Time Tagger X only	53
5.1.3	High Resolution Mode	54
5.2	Data connection	54
5.3	LEDs	55
5.3.1	Time Tagger X	55
5.3.2	Time Tagger Ultra	56
5.3.3	Time Tagger 20	57
5.4	Test signal	57
5.5	Synthetic input delay	57
5.6	Synthetic dead time	58
5.7	Event divider	58
5.8	Conditional Filter	58
5.9	Bin equilibration	58
5.10	Overflows	59
5.11	External Clock Input	59
5.12	Synchronization signals	60
5.13	FPGA link	60
5.14	General purpose IO (GPIO)	60

6	Software Overview	63
6.1	Graphical User Interfaces	63
6.2	Precompiled libraries and high-level language bindings	63
6.3	C++ API	63
7	Application Programming Interface	65
7.1	Examples	65
7.1.1	Measuring cross-correlation	65
7.1.2	Using virtual channels	66
7.1.3	Using multiple Time Taggers	66
7.1.4	Using Time Tagger remotely	67
7.2	The TimeTagger Library	68
7.2.1	Units of measurement	68
7.2.2	Channel numbers	68
7.2.3	Unused channels	68
7.2.4	Constants	68
7.2.5	Enumerations	68
7.2.6	Functions	71
7.2.7	Helper classes	75
7.3	TimeTagger classes	75
7.3.1	General Time Tagger features	76
7.3.2	Time Tagger hardware	81
7.3.3	The TimeTaggerVirtual class	92
7.3.4	The TimeTaggerNetwork class	93
7.3.5	Additional classes	95
7.4	Virtual Channels	96
7.4.1	Available virtual channels	96
7.4.2	Common methods	97
7.4.3	Coincidence	97
7.4.4	Coincidences	98
7.4.5	Combinations	99
7.4.6	Combiner	101
7.4.7	ConstantFractionDiscriminator	101
7.4.8	DelayedChannel	102
7.4.9	EventGenerator	103
7.4.10	FrequencyMultiplier	104
7.4.11	GatedChannel	104
7.4.12	TriggerOnCounter	105
7.5	Measurement Classes	107
7.5.1	Available measurement classes	107
7.5.2	Common methods	108
7.5.3	Event counting	110
7.5.4	Time histograms	115
7.5.5	Fluorescence-lifetime imaging (FLIM)	127
7.5.6	Phase & frequency analysis	136
7.5.7	Time-tag-streaming	146
7.5.8	Helper classes	155
7.5.9	Custom Measurements	157
8	In Depth Guides	159
8.1	Conditional Filter	159
8.1.1	Example configurations	159
8.1.2	Understanding the filtering mechanism	161
8.1.3	Setup of the Conditional Filter	164

8.2	Raw Time-Tag-Stream access	165
8.2.1	Dumping and post-processing	165
8.2.2	On-the-fly processing	166
8.3	Synchronization of the Time Tagger pipeline	166
8.4	FPGA link	167
8.4.1	Getting Started with SFP+	167
8.4.2	Using QSFP+	168
8.4.3	Modifying the reference design	168
9	Usage Statistics Collection	169
9.1	Contents of the usage statistics data	169
9.2	Ways of control	169
9.3	Time Tagger Lab diagnostics	170
10	Legal & Safety	171
10.1	<i>Time Tagger X</i> - Safety notice	171
10.1.1	Symbols	171
10.1.2	Operation environment	171
10.1.3	Electrical characteristics	172
10.1.4	Electrostatic-sensitive device	172
10.1.5	Equipment installation	172
10.1.6	Maintenance and repair	173
10.1.7	Disposal and recycling	174
10.1.8	Contact, support and service	175
11	Revision History	177
11.1	V2.17.4 - 17.07.2024	177
11.2	V2.17.2 - 02.07.2024	177
11.3	V2.17.0 - 22.04.2024	178
11.4	V2.16.2 - 28.06.2023	179
11.5	V2.16.0 - 05.06.2023	180
11.6	V2.15.0 - 06.03.2023	181
11.7	V2.14.0 - 23.12.2022	182
11.8	V2.13.2 - 22.11.2022	183
11.9	V2.12.4 - 09.11.2022	183
11.10	V2.12.2 - 04.10.2022	183
11.11	V2.12.0 - 01.09.2022	183
11.12	V2.11.0 - 22.04.2022	184
11.13	V2.10.6 - 16.03.2022	186
11.14	V2.10.4 - 23.02.2022	186
11.15	V2.10.2 - 31.12.2021	186
11.16	V2.10.0 - 22.12.2021	187
11.17	V2.9.0 - 07.06.2021	189
11.18	V2.8.4 - 04.05.2021	190
11.19	V2.8.2 - 26.04.2021	190
11.20	V2.8.0 - 29.03.2021	190
11.21	V2.7.6 - 26.04.2021	191
11.22	V2.7.4 - 19.04.2021	191
11.23	V2.7.2 - 22.12.2020	191
11.24	V2.7.0 - 01.10.2020	193
11.25	V2.6.10 - 07.09.2020	193
11.26	V2.6.8 - 21.08.2020	194
11.27	V2.6.6 - 10.07.2020	194
11.28	V2.6.4 - 27.05.2020	195

11.29 V2.6.2 - 10.03.2020	196
11.30 V2.6.0 - 23.12.2019	198
11.31 V2.4.4 - 29.07.2019	199
11.32 V2.4.2 - 12.05.2019	200
11.33 V2.4.0 - 10.04.2019	200
11.34 V2.2.4 - 29.01.2019	201
11.35 V2.2.2 - 13.11.2018	201
11.36 V2.2.0 - 07.11.2018	201
11.37 V2.1.6 - 17.05.2018	202
11.38 V2.1.4 - 21.03.2018	202
11.39 V2.1.2 - 14.03.2018	202
11.40 V2.1.0 - 06.03.2018	202
11.41 V2.0.4 - 01.02.2018	202
11.42 V2.0.2 - 17.01.2018	203
11.43 V2.0.0 - 14.12.2017	203
11.44 V1.0.20 - 24.10.2017	203
11.45 V1.0.6 - 16.03.2017	204
11.46 V1.0.4 - 24.11.2016	205
11.47 V1.0.2 - 28.07.2016	206
11.48 V1.0.0	206
11.49 Channel Number Schema 0 and 1	206

Index	207
--------------	------------

GETTING STARTED

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

First, please install the most recent driver/software, including graphical user interfaces, libraries, and examples for C++, Python, .NET, C#, LabVIEW, Matlab, and Mathematica.

Time Tagger software download

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

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

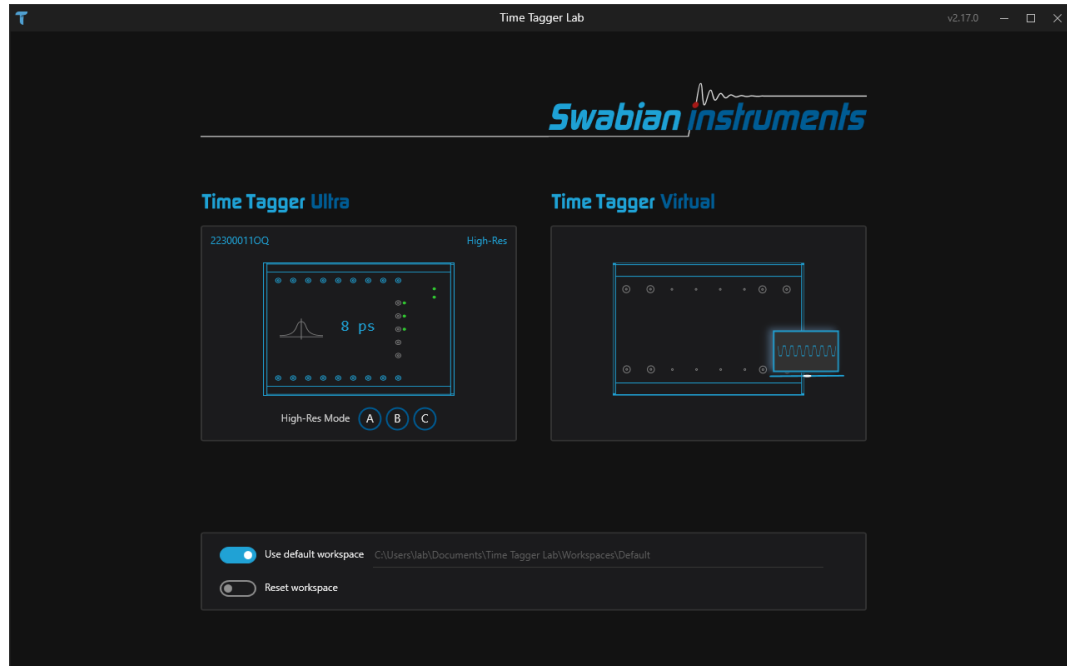
Additional information about the hardware, API, etc. can be found in the menu bar on the left and on our main website: <https://www.swabianinstruments.com/time-tagger/>.

If you are using Linux, please take a look at the [Linux](#) section.

1.1 Get familiar with your Time Tagger

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

1. Run the Graphical User Interface, *Time Tagger Lab* or the *Web Application*, to play with your Time Tagger interactively. The examples below will allow you to experience basic data acquisition. Play with the Time Tagger settings to see their effects on typical measurements.
2. Check out the *Application Programming Interface* chapter. It gives you a detailed overview of all Time Tagger features and data processing classes. Check out the following sections to get started using the *Time Tagger* software library in the programming language of your choice.
3. Study the code examples in the `.\examples\<language>` folders of your Time Tagger installation.



Time Tagger Lab 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. On the welcome screen, click on the panel representing your desired device. The Home screen of your Time Tagger shows up.
2. On the top of the Home screen, you can toggle between the basic display you are currently looking at and the Detailed View. Activate the Detailed View.
3. In the upper panel of the Detailed View, you can change settings for every input. Activate the internal test signal on inputs 1 and 2 by checking the boxes on the rightmost column.
4. Go back to the basic display by deactivating Detailed view. The count rate displays (cps) show a test signal count rate of 800 to 900 kHz, and the trigger level setting is replaced by the label test signal.

With the activated test signal, we can start our first measurement.

1. Click on **Open creator** (F2) in the **Measurements** panel on the left. The creator gives you an overview of all available measurements.
2. Pick **Bidirectional histogram - Correlation**. Now you can read through a detailed description and change the initial settings of your measurement.
3. Set **Reference channel** to 1 and **Click channel** to 2. Click **Add measurement**.
4. The measurement graph is showing up. To start the measurement, click the **Play** button next to the graph.
5. A Gaussian peak should be displayed. You can zoom in using the controls on the plot. If the resolution is not sufficient, try a smaller **Bin width** in the measurement settings.
6. The detection jitter of a single channel is $\frac{1}{\sqrt{2}}$ times the standard deviation of this two-channel measurement (the FWHM (Full-width at half-maximum) of the Gaussian peak is 2.35 times its standard deviation).

You have just verified the time resolution (detection jitter) of your Time Tagger.

1.2.2 Web Application

The Web Application is the traditional provided GUI (Graphical User Interface) for Windows and Linux for quick measurements. On Windows systems, we recommend switching to *Time Tagger Lab* today.

1. Download and install the most recent [Time Tagger software](#) from our downloads site.
2. Start the Time Tagger Application from the Windows start menu.
3. The Web Application should show up in your browser.

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

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

1. Click Add TimeTagger, click Init (select resolution if available) on any of the available Time Taggers
2. Click Create measurement, look for Bidirectional Histogram (Class: Correlation) and click Create next to it.
3. Select Rising edge 1 for Channel 1 and Rising edge 2 for Channel 2.
4. Set Binwidth to 10 ps and leave Number of data points at 1000, click Initialize.

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

1. On the top left, click on the settings wheel next to Time Tagger.
2. On the far right, check Test signal for channels 1 and 2, click Ok.
3. A Gaussian peak should be displayed. You can zoom in using the controls on the plot.
4. The detection jitter of a single channel is $\frac{1}{\sqrt{2}}$ 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.

1.3 Programming languages

1.3.1 Python

1. Make sure that your *Time Tagger* device is connected to your computer and the *Time Tagger* Web Application (especially the server window) is closed.
2. Make sure the *Time Tagger* software and a Python distribution (on Windows OS we recommend [Anaconda](#)) are installed.
3. Open a command shell and navigate to the `.\examples\python\1-Quickstart` folder in your *Time Tagger* installation directory
4. Start an **ipython** shell with plotting support by entering `ipython --pylab`
5. Run the **hello_world.py** script by entering `run hello_world`

The *hello_world* executes a simple yet useful measurement that demonstrates many essential features of the *Time Tagger* programming interface:

1. Connect your Time Tagger

2. Start the built-in test signal (~0.8 MHz square wave) and apply it to channels 1 and 2
3. Control the trigger level of your inputs - although it is not necessary here
4. Initialize a standard measurement (*Correlation*) in order to find the delay of the test signal between channel 1 and 2
5. How to control the delay of different inputs programmatically.

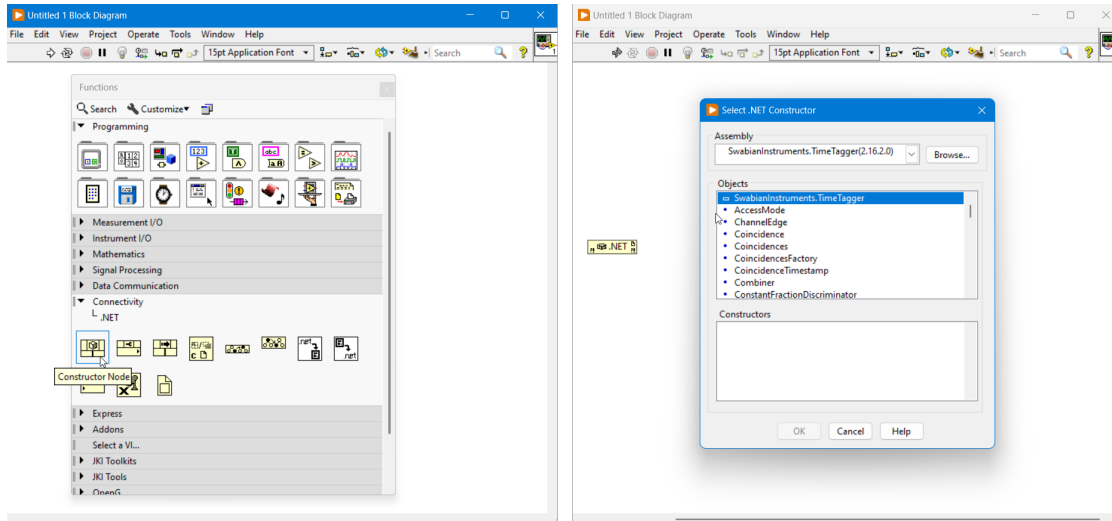
You are encouraged to open and read the `hello_world.py` file in an editor to understand what it is doing. With this basic knowledge, you can explore the other examples in the 1-Quickstart folder:

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

More details about the software interface are covered by the API documentation in the subsequent section

1.3.2 LabVIEW (via .NET)

In LabVIEW, you can access and program your Time Tagger through .NET interoperability.



A set of examples is provided in `.\examples\LabVIEW\` for LabVIEW 2014 and higher (32 and 64 bit).

1.3.3 Matlab (wrapper for .NET)

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

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

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

- static functions are available through the TimeTagger class
- all classes except for the TimeTagger, TimeTaggerNetwork, and TimeTaggerVirtual classes will have a TT prefix (e.g. TTCountrate) to prevent conflict with any variables/classes in your Matlab environment

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

1.3.4 Wolfram Mathematica (via .NET)

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

1.3.5 .NET

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

The following are important to note:

- Namespace: `SwabianInstruments.TimeTagger`
- the corresponding library `.\driver\xxx\SwabianInstruments.TimeTagger.dll` is registered in the Global Assembly Cache (GAC)

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

1.3.6 C#

A sample Visual Studio C# project provided in the `.\examples\csharp\Quickstart` directory covers the basics of how to use the Time Tagger .NET API. An example of creating ‘custom measurements’ is also included.

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

An ‘Example Suite’ is provided in the `.\examples\csharp\ExampleSuite` directory. ‘Example Suite’ is an interactive application that demonstrates various measurements that can be performed with the TimeTagger. Reference source code to setup and plot (with OxyPlot) each measurement is also provided within the application. Additionally, the application contains examples for creating and using ‘*Virtual channels*’, ‘*Filtering*’ and ‘*Accessing the raw time tags*’.

Note: Running the Example Suite requires ‘.NET Core 3.1 Desktop Runtime (v3.1.10)’.

1.3.7 C++

The provided Visual Studio C++ project can be found in `.\examples\cpp\`. Using the C++ interface is the most performant way to interact with the Time Tagger as it supports writing custom measurement classes with no overhead. But it is more elaborate compared to the other high-level languages. Please visit `.\documentation\Time Tagger C++ API Manual.pdf` for more details on the C++ API.

Note:

- the C++ headers are stored in the `.\driver\include\` folder
 - the final assembly must link `.\driver\xYZ\TimeTagger.lib`
 - the library `.\driver\xYZ\TimeTagger.dll` is linked with the shared v142 or newer Visual Studio runtime (`/MD`)
 - use `TimeTaggerD.lib` and `TimeTaggerD.dll` for the Visual Studio debug runtime (`/MDd`)
 - use `libTimeTagger.dll` and `libTimeTagger.a` for the MinGW C++ ABI for the [MINGW32](#) and [UCRT64 environment](#).
-

Debug and release builds: Performance issues with debug builds

For this topic, it is required to be a bit more precise on what is called a Debug build. Visual Studio default compiler flags for Debug are `/MDd /O0 /Zi` and for Release `/MD /O2 /DNDEBUG`.

- `/O0` vs `/O2`

This is the general optimization level. `/O0` means no optimization, so every instruction is surrounded by `load_from_memory` and `store_to_memory`. This is a huge waste of CPU resources, but it guarantees that every local variable can be inspected at every time. We suggest to use the default optimization level `/O2` by default and to overwrite it only for required methods using pragmas: [MSVC optimize pragma](#).

- **/MDd vs /MD**

This flag selects which (incompatible) runtime C++ ABI you want to use for the whole project. The debug runtime MDd has additional fields and checks for e.g. better range checking. It is good practice to run your code with them at least once before each commit. As they are incompatible with each other, ALL linked C++ libraries MUST use the same runtime. This flag is the only difference between our debug TimeTaggerD.dll and non-debug TimeTagger.dll library. We suggest to use MD even for debug builds, else the debug runtime will degenerate the overall performance also in the internal Time Tagger code.

- **/Zi**

This flag tells the compiler to generate a *.pdb symbol file. This is used by the debugger to map e.g. the assembly to the C++ code and the stack to the local variable. As this has no performance overhead, always use this flag for all internal builds.

- **/DNDEBUG**

The macro NDEBUG is the C way to disable assert(). They are usually good at catching same errors without much overhead. We also like to recommend to switch your compiler to CLANG and use its undefined behavior sanitizer, the address sanitizer or the thread sanitizer. Microsoft announced it a bit ago that MSVC now has the address sanitizer, which finds out-of-bounds accesses, use-after-free and similar issues: [AddressSanitizer \(ASan\) for Windows with MSVC](#).

So in the end, it is not about just using Debug or Release. It is about which set of flags is used. Please select each of them carefully based on your target.

INSTALLATION INSTRUCTIONS

2.1 Windows

Time Tagger software requires Windows 10 or higher. We provide separate Windows installers for 32 and 64 bit systems.

2.1.1 Installation

1. Download the installer from our [downloads site](#).
2. Run the installer and follow the instructions.
3. Connect your Time Tagger.
4. Make sure that your computer is connected to the internet once you run the Time Tagger software. The software needs to request its license from our server. Once the license is transferred, no internet connection is required anymore.

2.2 Linux

2.2.1 Installation

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

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

Graphical user interface (Web Application):

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

2.2.2 Known issues

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

2.2.3 Time Tagger with Python

- Install NumPy (e.g. `pip install numpy`), which is required for the Time Tagger libraries.
- The Python libraries are installed in your default Python search path: `/usr/lib/pythonX.Y/dist-packages/` or `/usr/lib64/pythonX.Y/site-packages/`.
- The examples can be found within the `/usr/lib/timetagger/examples/python/` folder.

You can compile a Python module for custom Python installations in the following way:

The source of the Python wrapper `_TimeTagger.cxx` is provided in `/usr/lib64/pythonX.Y/site-packages/`. For building the wrapper, the GNU C++ compiler and the development headers of Python and numpy need to be installed. The resulting `_TimeTagger.so` and the high-level wrapper `TimeTagger.py` relay the Time Tagger C++ interface to Python.

```
PYTHON_FLAGS="-python3-config --includes --libs`"
NUMPY_FLAGS="-I`python3 -c \"print(__import__('numpy').get_include())\"`"
TTFLAGS="-I/usr/include/timetagger -lTimeTagger"
CFLAGS="-std=c++17 -O2 -DNDEBUG -fPIC $PYTHON_FLAGS $NUMPY_FLAGS $TTFLAGS"

g++ -shared _TimeTagger.cxx $CFLAGS -o _TimeTagger.so
```

2.2.4 Time Tagger with C++

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

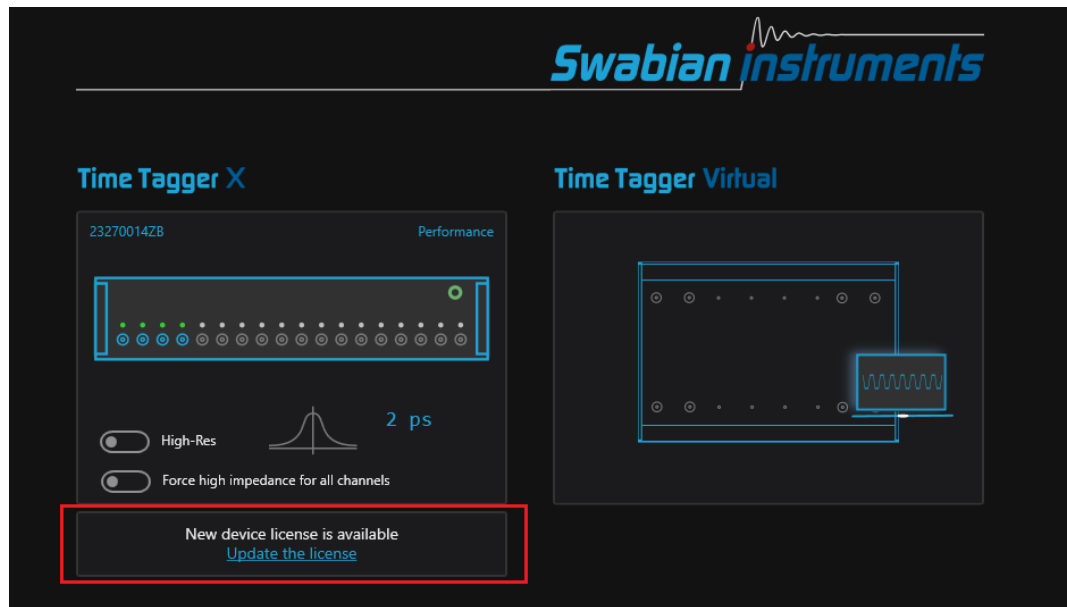
The C++ interface will likely also work on other distributions out of the box.

2.3 Hardware license upgrades

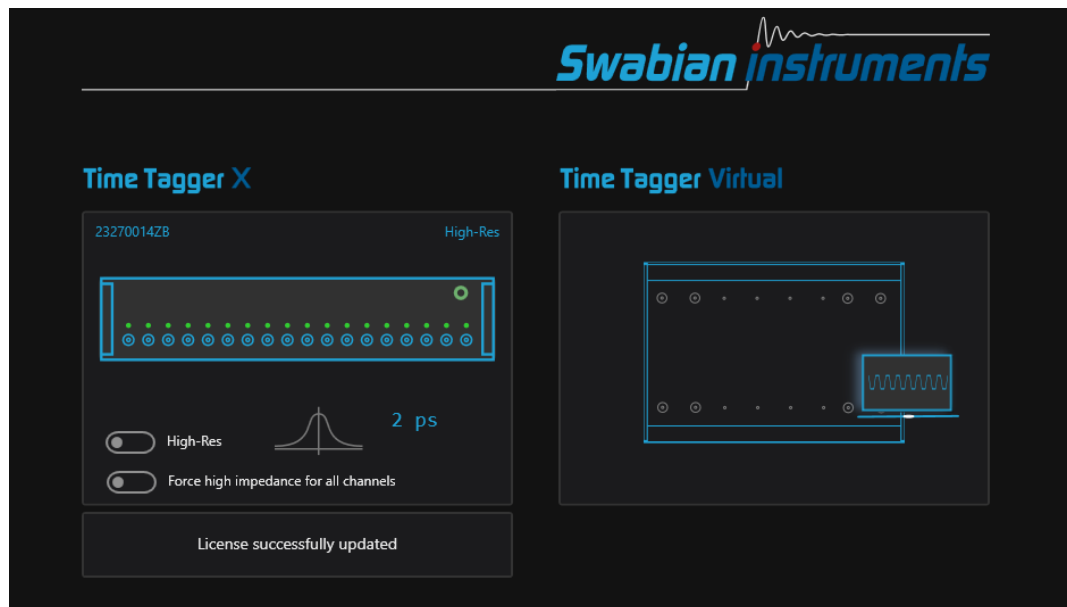
The Time Tagger Ultra and Time Tagger X have a hardware license that can be upgraded to activate additional channels and features. To upgrade your license, please get in touch with sales@swabianinstruments.com. After purchasing a license upgrade, the new license becomes available within a few minutes. You can install your new license using one of the graphical user interfaces. An active internet connection is needed once for the upgrade process.

2.3.1 Time Tagger Lab

1. Connect the Time Tagger to your PC
2. Start Time Tagger Lab
3. If a new license is available, you will see the following:

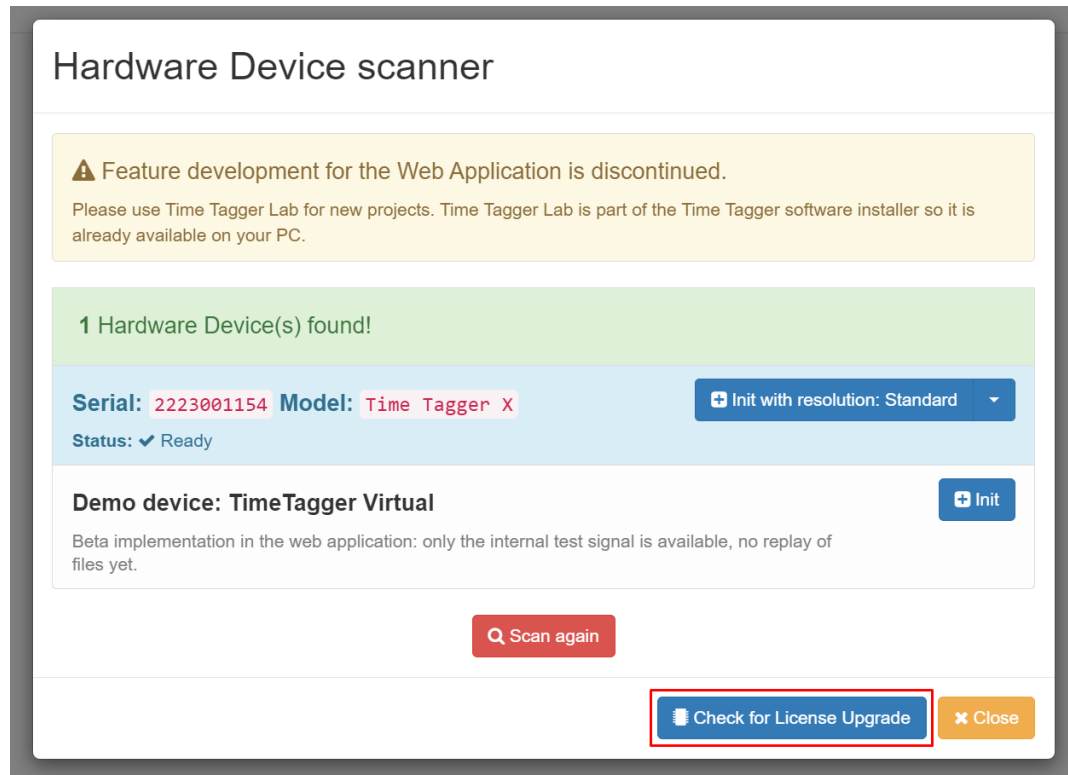


4. Click on "Update the license". The new license will be applied immediately and you will see:

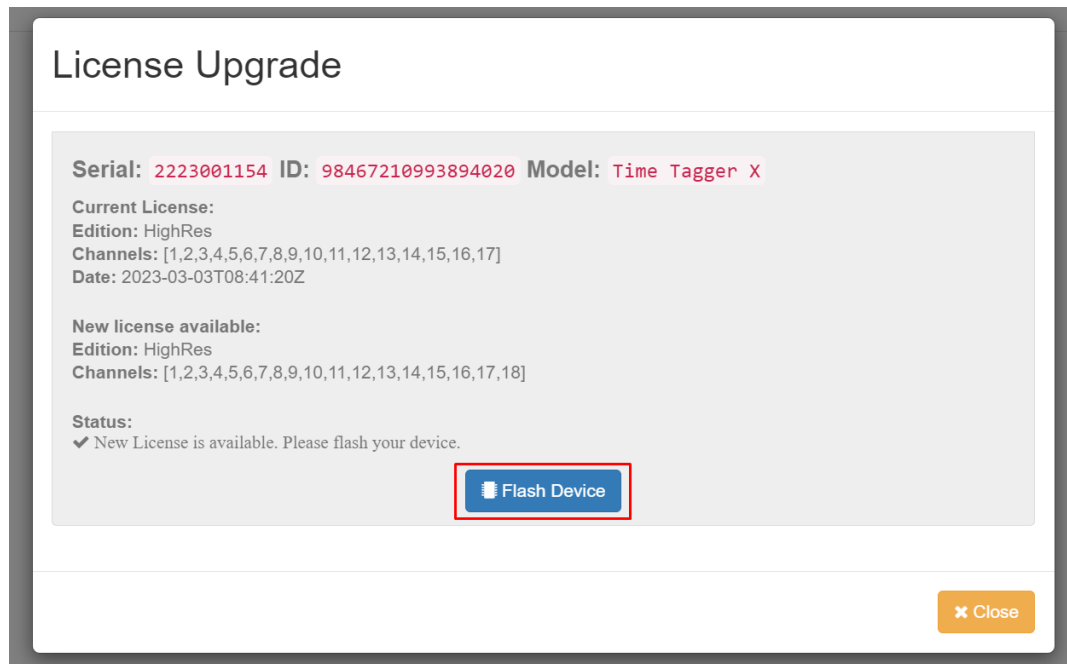


2.3.2 Web Application

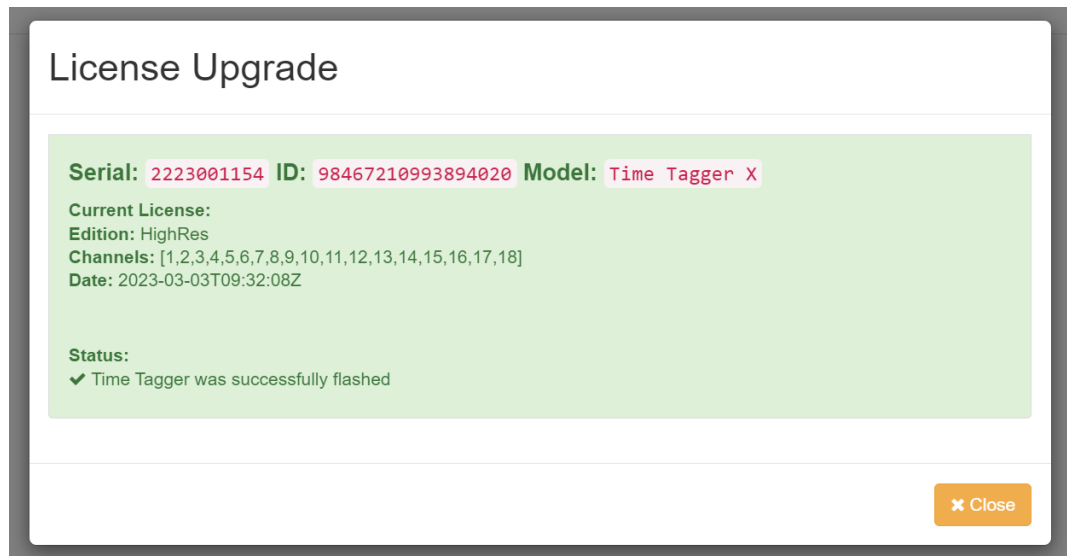
1. Connect the Time Tagger to your PC
2. Start the Time Tagger WebApplication
3. Click on “Add Time Tagger” in the top left corner
4. You should see your connected Time Tagger and at the bottom the button “Check for License Upgrade” (the serial number should be different)



5. Click on “Check for License Upgrade”
6. If a new license is available, you will see the following:



7. If you want to activate the new license, please click “Flash Device” and wait until the process is complete. You will see the following message:



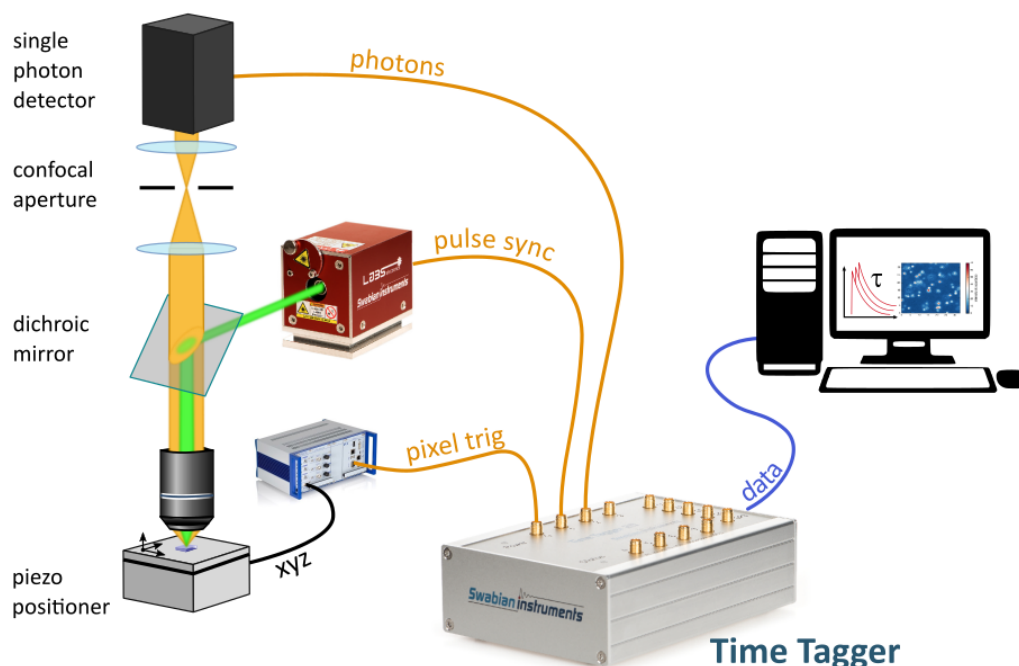
TUTORIALS

3.1 Confocal Fluorescence Microscope

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

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

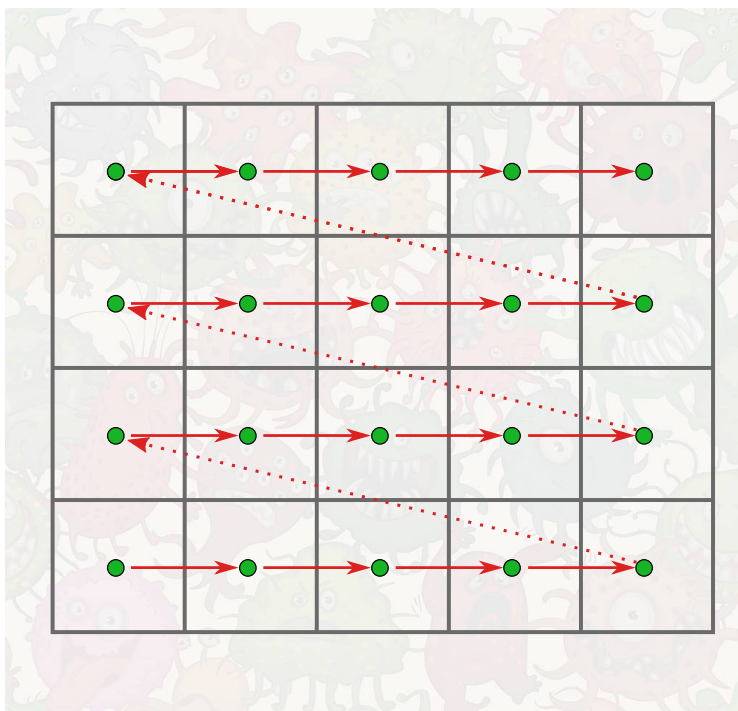
The following drawing shows a typical confocal fluorescence microscope setup.



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

Image from a raster scan

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



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

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

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

3.1.1 Time Tagger configuration

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

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

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

Now let's connect to the Time Tagger.

```
tt = createTimeTagger()
```

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

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

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

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

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

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

3.1.2 Intensity scanning microscope

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

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

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

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

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

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

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

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

script can continue immediately after starting the scan.

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

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

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

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

3.1.3 Fluorescence Lifetime Microscope

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

The *TimeDifferences* measurement calculates the time differences between laser and photon pulses and accumulates them in a histogram for every pixel. The measurement class constructor requires imaging and timing parameters, as shown in the following code snippet.

```
nx_pix = 300 # Number of pixels along x-axis
ny_pix = 200 # Number of pixels along y-axis
binwidth = 50 # in picoseconds
n_bins = 2000 # number of bins in a histogram
n_pixels = nx_pix * ny_pix # number of histograms

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

Now we start the scanner and wait until the scan is completed. During the scan, we can read the current data and display it in real time.

```
while scanner.isScanning():
    counts = flim.getData()
    img3D = np.reshape(counts, n_bins, nx_pix, ny_pix) # Fluorescence image cube
```

(continues on next page)

(continued from previous page)

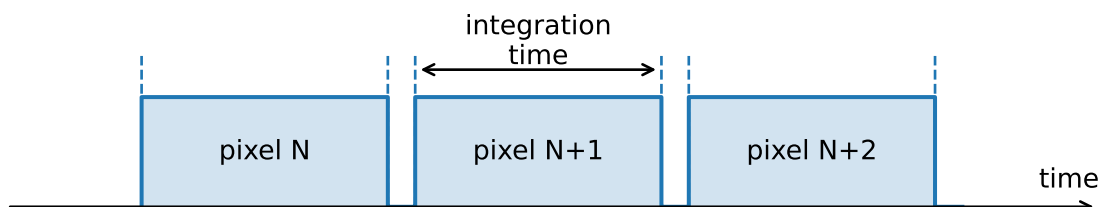
```
# User defined function that estimates fluorescence lifetime for every pixel
flimg = get_lifetime(img3D)

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

3.1.4 Alternative pixel trigger formats

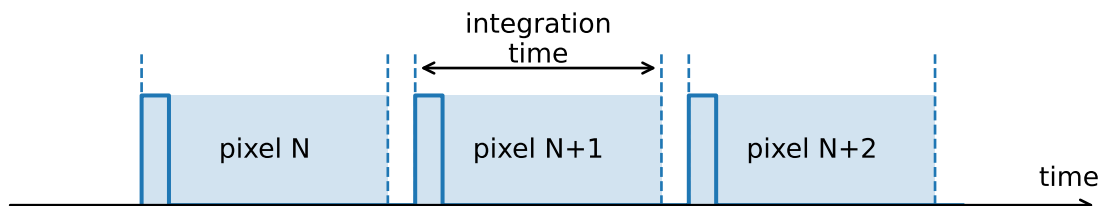
What if a piezo-scanner provides a different trigger signal compared to considered in the previous sections? In this section, we look into a few common types of trigger signals and how to adapt our data acquisition to make them work.

Pixel pulse width defines the integration time



The case when the pulse width defines the integration time has been considered in the previous subsections.

Pixel pulse indicates the pixel start



When a pixel pulse has a duration different from the desired integration time, we must define the integration time manually. One way would be to record all events until the next pixel pulse and rely on a strictly fixed pixel pulse period. Alternatively, we can create a well-defined time window after each pixel pulse, so the measurement system becomes insensitive to the variation of the pixel pulse period.

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

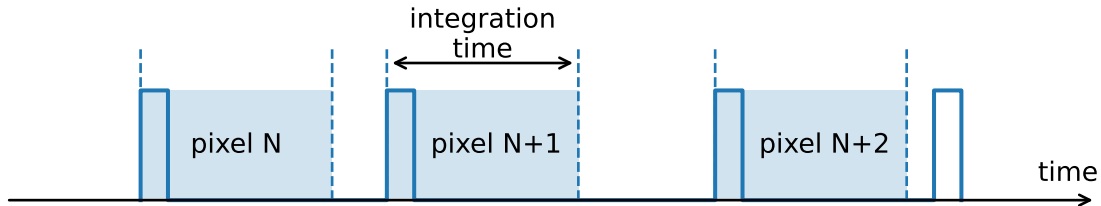
```
integr_time = int(1e10) # Integration time of 10 ms in picoseconds
delayed_vch = DelayedChannel(tt, PIXEL_START_CH, integr_time)
PIXEL_END_CH = delayed_vch.getChannel()

cbm = CountBetweenMarkers(tt, SPD_CH, PIXEL_CH, PIXEL_END_CH, n_pixels)
```

The approach with using [DelayedChannel](#) allows for a constant integration time per pixel even if the pixel pulses do not occur at a fixed period. For instance, in a raster scan, more time is required to move to the beginning of the next line (fly-back time) compared to the pixel time.

Warning: You have to make sure that pixel pulses do not appear before the end of the integration time for the previous pixel.

FLIM with non-periodic pixel trigger



In some cases, a scanner generates the pixel pulses with no strictly defined period. However, most scanning measurements require constant integration time for every pixel. Compared to [CountBetweenMarkers](#), the [TimeDifferences](#) measurement do not have a *PIXEL_END* marker and accumulate the histogram for every pixel until the next pixel pulse is received. If this behavior is undesired, or if your pixel pulses are not periodic, you will need to gate your detector to guarantee a constant integration time.

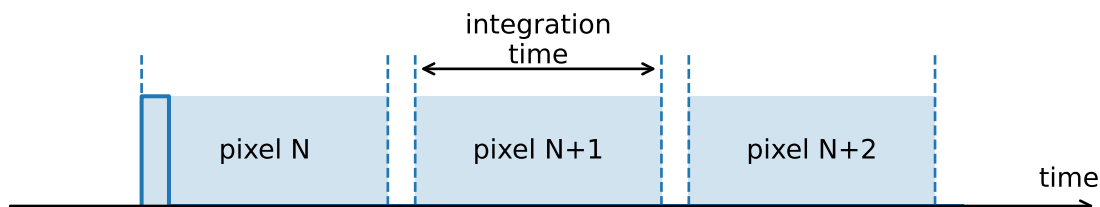
The Time Tagger library provides you with the necessary tools to enforce a fixed integration time when using the [TimeDifferences](#) measurement. Gating the detector events can be done with the [GatedChannel](#). The example code is provided below.

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

gated_vch = GatedChannel(tt, SPD_CH, PIXEL_START_CH, PIXEL_END_CH)
GATED_SPD_CH = gated_vch.getChannel()

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

Line pulse but no pixel pulses



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

```

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

LINE_START_CH = 3

# Pixels are equally spaced in time (constant speed)
pixel_start_times = numpy.linspace(0, line_duration, nx_pix, dtype='int64')
# Pixel integration time is constant
pixel_stop_times = pixel_start_times + integr_time

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

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

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

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

```

(continues on next page)

(continued from previous page)

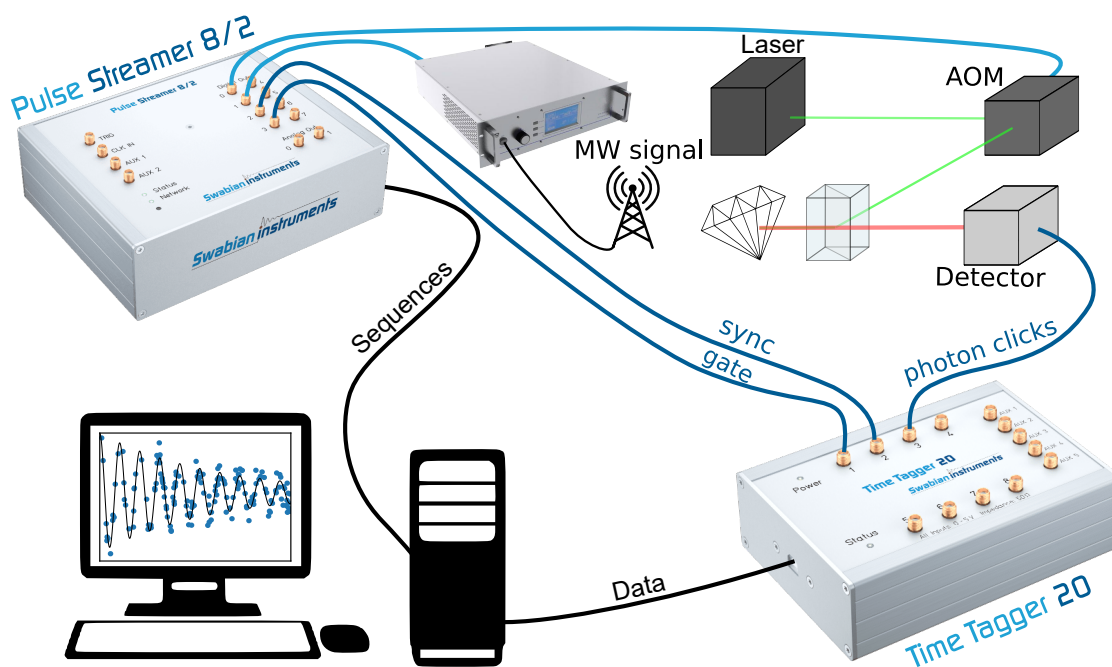
```
n_histograms=n_pixels
```

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

3.2 Optically Detected Magnetic Resonance

Optically Detected Magnetic Resonance (ODMR) is a spectroscopic technique used to study electron spins in materials. It involves applying sequences of optical and microwave pulses to manipulate and probe the spin states of electrons. By analyzing the response of the material's photoluminescence to these pulses, valuable information about the spin properties and dynamics can be obtained. This technique is particularly useful for studying spin-based quantum technologies and materials such as diamond NV centers.

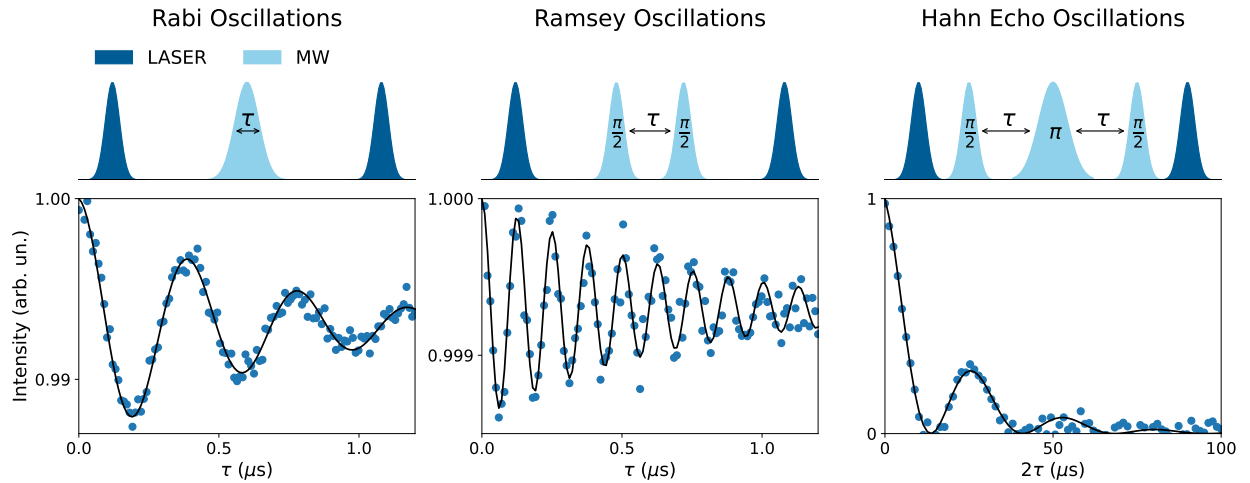
This tutorial shows how to set up a pulsed ODMR experiment using our Time Taggers and Pulse Streamer. Pulsed ODMR offers distinct advantages over continuous mode, including enhanced contrast and reduced sensitivity to laser power fluctuations. Leveraging Time Taggers provides precise timing control for accurate data acquisition and analysis, making it ideal for probing fast spin dynamics with high resolution.



3.2.1 Creation of optical and microwave pulse patterns

In this tutorial we consider a spin-1 system. In most sequences, an initial optical pulse serves to initialize the system, e.g., preparing it in state $|m_s = 0\rangle$. Following this initialization, a series of microwave (MW) pulses and free evolution periods manipulate the spin state. Finally, a second optical pulse interrogates the spin state projection along the $(|m_s = 0\rangle, |m_s = +1\rangle)$ basis, often via optical emission or absorption intensity, while also resetting the system for subsequent measurements. Various pulse sequences are commonly employed for different measurements:

1. **Rabi Oscillation Sequence:** In this sequence, a single MW pulse of variable duration is applied, causing the spin to oscillate between the $|m_s = 0\rangle$ and $|m_s = +1\rangle$ states. The amplitude of the Rabi oscillations provides information about the energy splitting between the two states. From this measurement the duration of π - and $\pi/2$ -pulses can be inferred as half and quarter periods of the oscillations, respectively.
2. **Ramsey Sequence:** This sequence involves two $\pi/2$ MW pulses separated by a variable delay. The delay time determines the phase relationship between the two pulses, affecting the interference pattern observed in the spin state. Analysis of the Ramsey fringes allows for precise measurement of the spin coherence time T_2^* .
3. **Hahn Echo Sequence:** In this sequence, a $\pi/2$ -pulse is followed by a delay time and then a π -pulse. The second pulse flips the spin state, and the delay allows for dephasing to occur. The echo signal observed after the $\pi/2$ -pulse provides information about the spin dephasing time T_2 .



To create any of these sequences, we make use of the Sequence class of our Pulse Streamer 8/2. This class allows to independently set pulse patterns on each channel. A pulse pattern is represented by a tuple in the form (duration, level). The duration is always specified in nanoseconds and the level is either 0 or 1 for digital output.

First of all, we connect to the Pulse Streamer.

```
# Change the following line to use your specific Pulse Streamer IP address
ip_hostname='169.254.8.2'
pulser = PulseStreamer(ip_hostname)
```

Next, we declare the channels names, and all the relevant parameters for creating a sequence to measure Rabi Oscillations and quantifying the duration of a π -pulse. To achieve this, we need to create the following patterns:

1. A pattern to drive the laser, enabling it to emit optical pulses for system initialization and readout.
2. A pattern to drive a high isolation microwave switch for gating the microwave pulses.
3. A pattern for synchronization purposes.
4. A pattern to measure the incoming photons during the readout pulse for time-gated analysis.

```

# Channel names
optical_ch = 0
mw_ch = 1
sync_ch = 2
gate_ch = 3
# Digital levels
HIGH=1
LOW=0

# Change the values according to your experiment
period = 8000 # period of each pattern
init_pulse_dur = 2000 # width of initialization optical pulses
readout_pulse_dur = 300 # width of readout optical pulses
pause = 1000 # time between a readout and next initialization pulse

```

We define then a set of values over which the MW pulse duration (τ) is swept, to ultimately build the desired pulse patterns.

```

tau = np.arange(100, 2000, 100)

# For each selected MW pulse width, the experiment is repeated multiple times
n_loops = 1000

# Optical pattern
optical_patt = [(init_pulse_dur, HIGH), (period-init_pulse_dur-readout_pulse_dur-pause, LOW),
                (readout_pulse_dur, HIGH), (pause, LOW)]*n_loops*len(tau)

# MW pattern
# Initialize the MW pulse pattern
mw_patt = []
for ti in tau:
    mw_patt.extend([(init_pulse_dur, LOW), (ti, HIGH), (period-init_pulse_dur-ti, LOW)]*n_loops)

# Pulse pattern that marks a new value of a MW pulse duration
sync_patt = [(10, HIGH), (period*n_loops-10, LOW)]*len(tau)
# Add last sync pulse to the pattern to mark the end of the acquisition
sync_patt.extend([(10, HIGH)])

# Pattern for gating detector clicks
gate_patt = [(period-readout_pulse_dur-pause, LOW), (readout_pulse_dur, HIGH),
             (pause, LOW)]*n_loops*len(tau)

# Set channels using class Sequence
seq = Sequence()
seq.setDigital(optical_ch, optical_patt)
seq.setDigital(mw_ch, mw_patt)
seq.setDigital(sync_ch, sync_patt)
seq.setDigital(gate_ch, gate_patt)

# Display your sequence

```

(continues on next page)

(continued from previous page)

```
seq.plot()
```

3.2.2 Signal generation and detection

To perform ODMR measurements, we connect the Time Tagger and declare the channels used. Here, we can also adjust the hardware settings for each channel.

```
tagger = createTimeTagger()
gate = 1
sync = 2
detector = 3
```

Then, we need to filter the incoming detector clicks revealed by the Time Tagger according to the generated gate pattern. This can be done at the software level by employing the virtual channel *GatedChannel*. The rising and falling edges of the gate signal are used to open and close each time gate, respectively.

```
gated_detector_vch = GatedChannel(tagger, detector, gate, -gate)
# Get the channel number that will be used in the CBM measurement
gated_detector = gated_detector_vch.getChannel()
```

Next, we set up a *CountBetweenMarkers* measurement to count the filtered events on a detector channel between sync events, that herald the change of the MW pulse width. The counts are accumulated in an array whose number of bins is equal to the number of τ values.

```
cbm = CountBetweenMarkers(tagger, gated_detector, sync, CHANNEL_UNUSED, len(tau))
cbm.start()
tagger.sync()
```

Now that the measurement is set up, the sequence can be run by the Pulse Streamer and the events counted by the Time Tagger. We run the sequence once

```
n_runs = 1
final = OutputState.ZERO()
pulser.stream(seq, n_runs, final)
```

and we collect the data

```
ready = False

# Periodically get data until the CountBetweenMarkers completes the acquisition
while ready is False:
    time.sleep(.2)
    # Check if the measurement is ready
    ready = cbm.ready()
    counts = cbm.getData()
    # You may add progress visualization code here
```

3.2.3 Sweeping modes

There are different ways to acquire and accumulate data in these pulsed ODMR experiments. We report here two protocols to collect events to visualize Ramsey oscillations.

In the first method, the interpulse delay τ is swept and the data are collected in a single acquisition step after executing the whole sequence. The example code is analogous to the one of the previous paragraph, except for the construction of the Ramsey sequence.

```
dur_pi = 200 # Change value according to the outcome of Rabi measurement
tau = np.arange(100, 1500, 100) # Interpulse time delay

mw_patt = []
for ti in tau:
    mw_patt.extend([(init_pulse_dur, LOW), (dur_pi/2, HIGH), (ti, LOW),
                    (dur_pi/2, HIGH), (period-ti-dur_pi-init_pulse_dur, LOW)]*n_loops)
```

In the second approach, we repeat the sequence for the same interpulse delay multiple times, as usual, and accumulate the data before changing the interpulse delay.

```
#The optical, the sync and gate patterns do not depend on the interpulse delay
optical_patt = [(init_pulse_dur, HIGH), (period-init_pulse_dur-readout_pulse_dur-pause,
    LOW),
                (readout_pulse_dur, HIGH), (pause, LOW)]*n_loops
sync_patt = [(10, HIGH), (period*n_loops-10, LOW)]
sync_patt.extend([(10, HIGH)])
gate_patt = [(period-readout_pulse_dur-pause, LOW), (readout_pulse_dur,HIGH), (pause,
    LOW)]*n_loops

seq = Sequence()
seq.setDigital(optical_ch, optical_patt)
seq.setDigital(sync_ch, sync_patt)
seq.setDigital(gate_ch, gate_patt)

counts = []

# For each interpulse delay, events on the detector channel
# between two sync marker events are accumulated
cbm = CountBetweenMarkers(tagger, gated_detector, sync, CHANNEL_UNUSED, 1)
tagger.sync()

mw_patt = []
for ti in tau:
    mw_patt = [ (init_pulse_dur, LOW), (dur_pi/2, HIGH), (ti, LOW),
                (dur_pi/2, HIGH), (period-ti-dur_pi-init_pulse_dur, LOW)]*n_loops

    seq.setDigital(mw_ch, mw_patt)

    cbm.start()

# Run the sequence
pulser.stream(seq, n_runs, final)
```

(continues on next page)

(continued from previous page)

```

ready = False

# Get data every while
while ready is False:
    time.sleep(.2)
    ready = cbm.ready()
    data = cbm.getData()

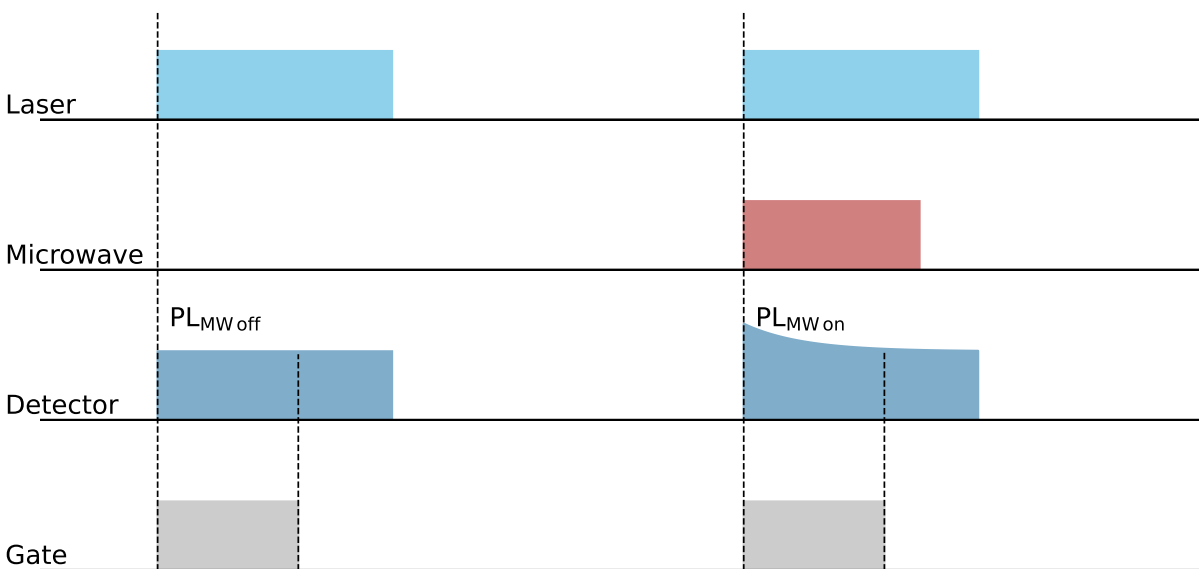
# Store the total counts for each interpulse delay into an array
counts.append(data)

```

3.2.4 ODMR contrast

Quantifying the ODMR contrast, defined as the differential photoluminescence signal between measurements with and without applying microwave radiation, is crucial for several reasons. It serves as a direct indicator of the efficiency of spin manipulation and the sensitivity of the measurement setup. High contrast values typically correspond to better signal-to-noise ratios, enabling more precise determination of resonance frequencies and spin relaxation times.

The schematics of this measurement is reported in the figure below. This trivial sequence is repeated for each different microwave frequency selected, to obtain in the end an ODMR spectrum.



For the data acquisitions, there are two possible implementations:

1. **A single CountBetweenMarkers measurement:** the rising and falling edges of the gate are used as *begin_channel* and *end_channel*, respectively, and $2N$ as *n_values*, where N is the number of frequencies to measure. In the output array, one gets, for each frequency, the counts while the MW is off in the even bins (0,2,4,6,...) and the counts while the MW is on in the odd bins (1,3,5,7,...).

```
cbm = CountBetweenMarkers(tagger, detector, gate, -gate, 2N)
```

2. **Two different CountBetweenMarkers measurements:** one collects counts when the MW frequency is on and one when the MW frequency is off. In this case MW indicator pulses are needed as markers. For the first *CountBe-*

tweenMarkers the rising and the falling edges of the MW indicator pulse are used as *begin_channel* and *end_channel*, respectively. On the contrary, for the second measurement the falling edge of the MW indicator pulse is used as *begin_channel* and the rising edge as *end_channel*. The gate signal should be used to filter the detector clicks at the software level using the the virtual channel *GatedChannel*, as described in the previous paragraph. This ensures the same acquisition time, for each frequency, when the microwave of and when it is off.

```
mw_ind = 4

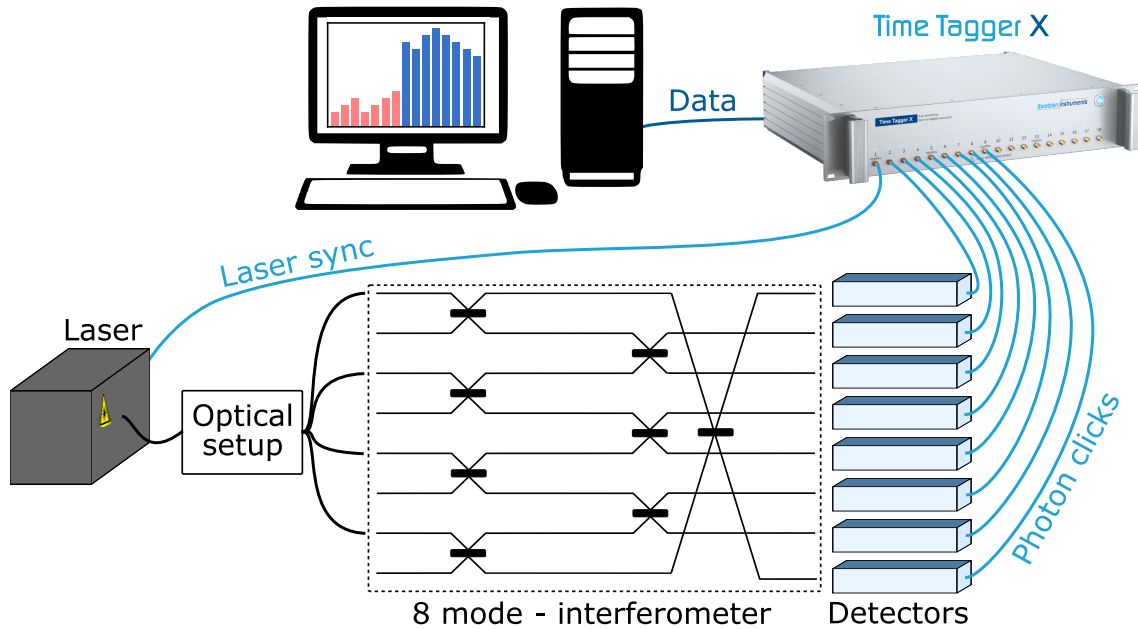
# Create a SynchronizedMeasurements instance that allows you to process the same tags
synchronized = SynchronizedMeasurements(tagger)
sync_tagger_proxy = synchronized.getTagger()

# Accumulate counts when the MW is on
cbm_mw_on = CountBetweenMarkers(sync_tagger_proxy, gated_detector, mw_ind, -mw_ind, N)

# Accumulate counts when the MW is off
cbm_mw_off = CountBetweenMarkers(sync_tagger_proxy, gated_detector, -mw_ind, mw_ind, N)
```

3.3 Measuring Coincidences

Quantum interference lies at the heart of photon-based technologies, such as photonic quantum computing, quantum metrology, and quantum networks. One of the key requirements for high-fidelity operations in these technologies is the indistinguishability of photons, a key property of quantum states. The level of indistinguishability between two photons is typically determined by measuring the Hong–Ou–Mandel (HOM) interference visibility: if the photons are identical and enter separately a balanced beam splitter, they will always exit the beam splitter together in the same output mode. If a detector is set up on each of the outputs, then the photons cannot be separately detected by the two detectors simultaneously. Thus, coincidence measurements enable the detection and analysis of correlated events, notably the simultaneous detection of entangled photons. When more than two photons are involved in a multiphoton quantum interference experiment, the interference capability extends beyond the pairwise distinguishability, and generalizations of the HOM effect to the many-particle case have been recently proposed. One method to quantify the n -photon indistinguishability relies on a cyclic multiport interferometer with $N = 2n$ optical modes, composed of $2n$ beam splitters placed along two cascaded layers. The quantum interference pattern, resulting from the injection of n indistinguishable photons into the $2n$ ports-interferometer and obtained through a multiphoton coincidence detection, is a direct measurement of the n -photon indistinguishability.



This tutorial shows how to set up a measurement to count coincidence events between groups of input channels. In our example, we count fourth-order coincidences between four photons injected into an integrated eight-mode cyclic interferometer.

3.3.1 Time Tagger configuration

To perform our coincidence-counting experiment, we first connect the Time Tagger and select the channels used.

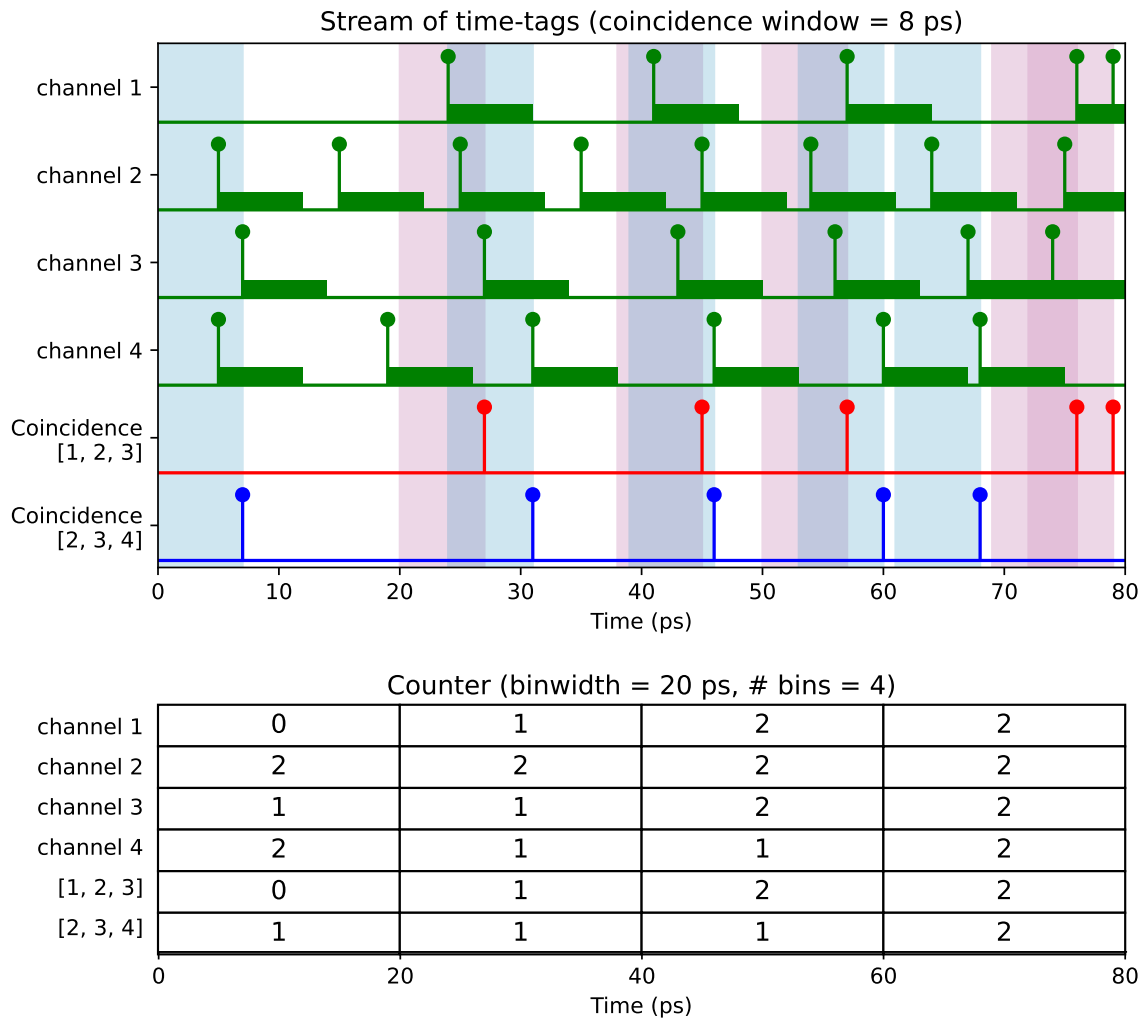
```
tagger = createTimeTagger()
input_channels = tagger.getChannelList(ChannelEdge.Rising)[:8]
```

The Time Tagger hardware allows you to specify a trigger level voltage for each input channel. The default trigger level is 0.5 V. In our example, we set the trigger level to 0.3 V for all the channels.

```
for ch in input_channels:
    tagger.setTriggerLevel(ch, 0.3)
```

3.3.2 Coincidence-counting

Our protocol for counting coincidence events is schematically illustrated in the figure below. For clear visualization, the diagram considers coincidences for two groups of three channels among four input channels used. First, we define a time window as the largest time difference between events on the input channels to be considered as a coincidence (size of rectangular shaded colored areas). Next, we generate new streams of tags, which correspond to coincidence events occurring for each group of input channels (Coincidence [1, 2, 3] and Coincidence [2, 3, 4]). Finally, we count the events on each channel (bottom panel).



Our method can be implemented by creating the virtual channel *Coincidences*, which detects coincidence clicks on multiple channel groups within the given coincidenceWindow. In this regard, it is good to remember that this is a software-defined channel; therefore, it receives time-tags from the physical input channels and generates a new data stream in the PC. The Time Tagger does not detect coincidences onboard the hardware.

In our example, we consider fourth-order coincidence events from eight input channels.


```

order = 4
groups = list(itertools.combinations(input_channels, order))

coincidences_vchannels = Coincidences(tagger, groups, coincidenceWindow=100)

```

Finally, the physical input channels and the *Coincidences* virtual channel are fed into the *Counter* measurement class.

```

#Generate a list including input and virtual channels
channels = [*input_channels, *coincidences_vchannels.getChannels()]

counting = Counter(tagger, channels, 1e10, 300)

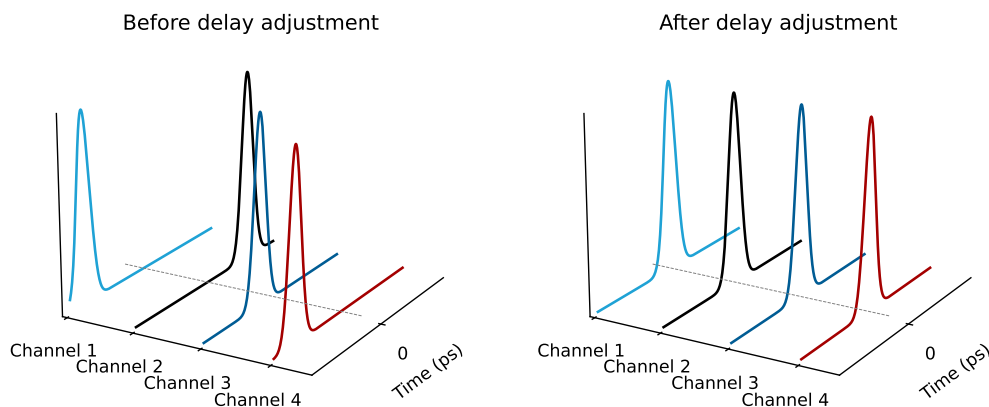
measurementDuration = 10e12 # 10 s
counting.startFor(measurementDuration)
counting.waitUntilFinished()

index = counting.getIndex()
counts = counting.getData()

```

3.3.3 Delay adjustment for coincidence detection

When detecting coincidence events, it is essential to have signals aligned in time. Delays between channels inevitably arise from experimental conditions, such as different optical paths or cable lengths, and inherent delays in the detectors. Such delays can be compensated in the Time Tagger by providing a proper input delay.



If the signals are sufficiently correlated, the best way to quantify the time misalignment between the channels is to perform multiple *Correlation* measurements. The peak of the *Correlation* curve between two channels is centered at the relative delay. This value is employed a posteriori to compensate the time misalignment between the two channels. In our example, we keep the first channel as a reference and alternatively measure the *Correlation* between it and all the others. Finally, we align all the signals in time, using `setInputDelay()` method.

```
# Set input delays to 0, otherwise the compensation result will be incorrect.
for ch in input_channels:
    tagger.setInputDelay(ch, 0)

# Create SynchronizedMeasurements to operate on the same time tags.
sm = SynchronizedMeasurements(tagger)

#Create Correlation measurements
corr_list = list()
for ch in input_channels[1:]:
    corr_list.append(
        Correlation(sm.getTagger(), input_channels[0], ch, binwidth=1, n_bins=5000)
    )

# Start measurements and accumulate data for 1 second
sm.startFor(int(1e12), clear=True)
sm.waitUntilFinished()

# Determine delays
delays = list()
for corr in corr_list:
    hist_t = corr.getIndex()
    hist_c = corr.getData()
    #Identify the delay as the center of the histogram through a weighted average
    dt = np.sum(hist_t * hist_c) / np.sum(hist_c)
    delays.append(int(dt))

print("Delays:", delays)

# Compensate the delays to align the signals
for ch, dt in zip(input_channels[1:], delays):
    tagger.setInputDelay(ch, dt)
```

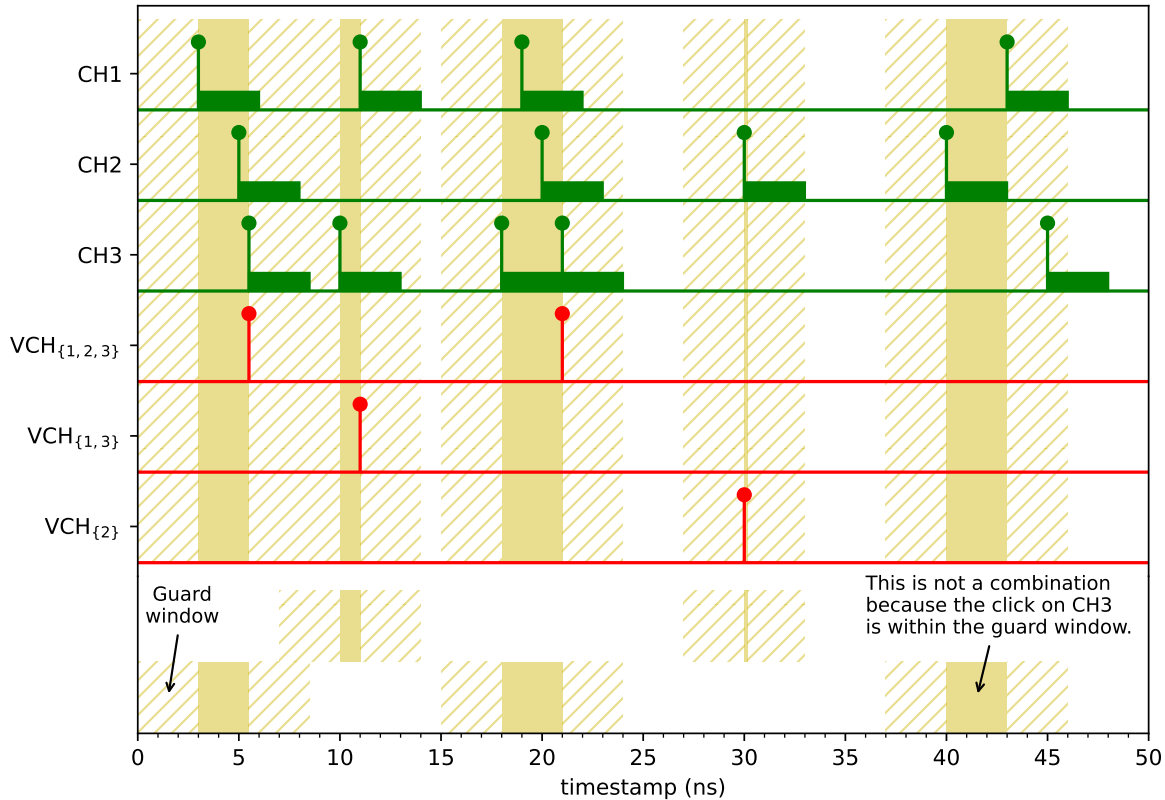
You can check that the delays are properly compensated by repeating the *Correlation* measurements and verifying that the histograms are centered at zero.

3.3.4 Exclusive coincidences using Combinations virtual channel

The indistinguishability of the input photons in a quantum interferometry experiment results in a large number of forbidden output configurations, among the possible many-particle states. This can be viewed as the generalization of the original two-photon HOM effect extended to the case of multiple ports and photons. In this regard, it turns out to be beneficial to consider coincidence measurements with specific restrictions. For instance, excluding certain coincidences based on the occurrence of events happening right before or after them can be important e.g., to eliminate the effect of background noise, hence for refining quantum state tomography or interference studies.

The best approach to achieve this goal is to employ the virtual channel *Combinations*. It detects coincidence clicks on all possible $(2^N - 1)$ channel subgroups from a given number (N) of channels, when no additional events occur within two guard windows, one preceding the first event starting the coincidence, the other following the last event concluding the coincidence. We report below a representative sketch of *Combinations* virtual channel given three input channels.

It should be noted that the combination of events on channels 1, 2, 3 results in the generation of a timestamp exclusively on the virtual channel $VCH_{\{1,2,3\}}$ and not on $VCH_{\{1,3\}}$ or $VCH_{\{2\}}$. This distinction, however, does not apply to the virtual channel *Coincidences*, as the coincidences between channels 1, 2, and 3 constitute a subset of coincidences between



1 and 3. Contextualizing this to the quantum interferometry experiment with four input photons and eight output ports, coincidences of order higher than four, made possible only by dark counts, are discarded on the fourth-order combinations. This is true if all eight channels are input to the *Combinations* virtual channel.

In the following minimal example, we demonstrate how to obtain the virtual channel numbers for fourth-order combinations, that can be input to the *Counter* measurement class. The goal of the example is to ensure that each combination includes at least one number from each of the sets {1, 2}, {3, 6}, {4, 5}, and {7, 8}.

```
# Create the Combinations virtual channel
combinations_vchannels = Combinations(tagger, input_channels, window_size=100)

#Define sets of inclusion
set1 = {1, 2}
set2 = {3, 6}
set3 = {4, 5}
set4 = {7, 8}

# Filter combinations of four channels groups based on conditions
filtered_combinations = [list(comb) for comb in groups
    if any(num in set1 for num in comb)
    and any(num in set2 for num in comb)
    and any(num in set3 for num in comb)
    and any(num in set4 for num in comb)]

# Create a list with the virtual channels monitoring the combinations
virtualchannelsnumber = []
```

(continues on next page)

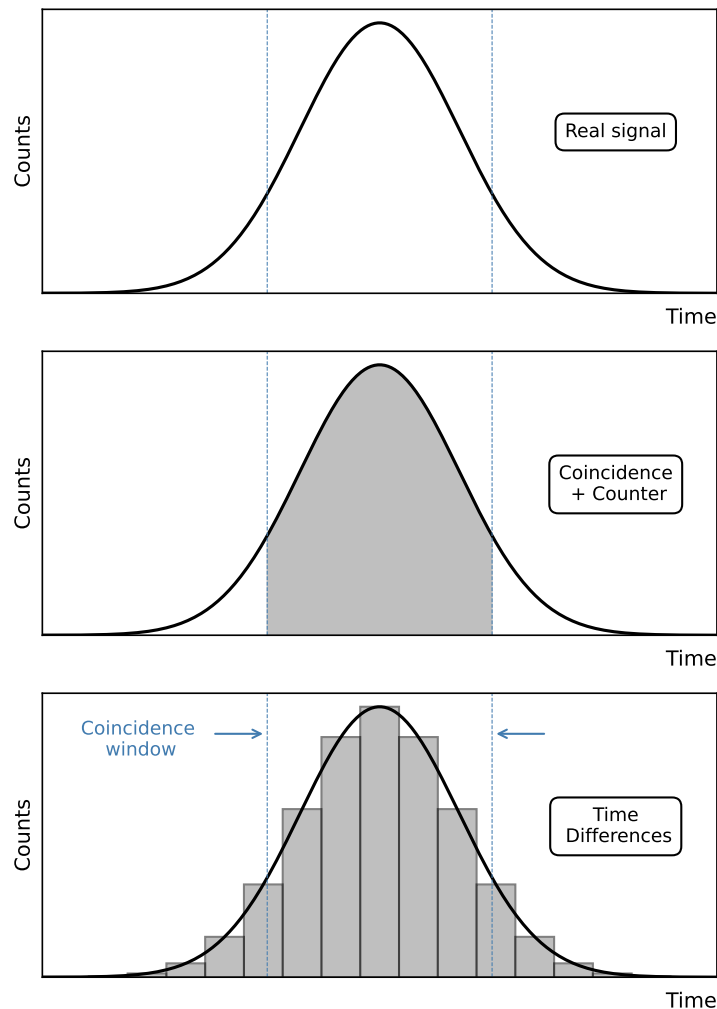
(continued from previous page)

```
for group in filtered_combinations:
    virtualchannelsnumber.append(combinations_vchannels.getChannel(group))
```

3.3.5 Coincidence-counting vs Correlation Measurement

Let us now assume in the following discussion to have only two channels. In this scenario, a coincidence-counting experiment could be also performed using other measurement classes, such as *Correlation*. The latter is a multi start, multi stop measurement, which means that every start click is correlated with every stop click as far they are within the *Correlation* time span. This implies that *Correlation* takes into account both positive and negative time differences. On the other hand, the “*ConstantFractionDiscriminator + Counter*” approach does not discriminate the order of the clicks, as the virtual channel detects an event whenever the time tags on the two channels are separated within the coincidenceWindow, regardless of whether this time difference is negative or positive.

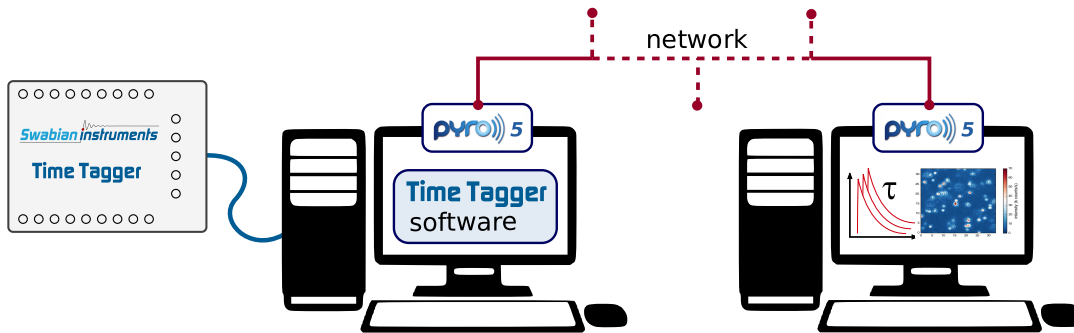
Coincidence counts can be retrieved by summing up a few bins of the *Correlation* histogram around zero time difference, depending on the size of the defined coincidence window. However, there is one point to consider for getting the correct number of coincidences using *Correlation* measurement class. Please see the figure below.



When summing the bins of the *Correlation* together to get coincidence counts, the bin alignment must be taken into

account, to get consistent results from the two analyses. It is indeed important to properly choose the binsize in *Correlation* such that the edges of the two outer bins align with the coincidenceWindow.

3.4 Remote Time Tagger with Python



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

In the following, we want to demonstrate two ways of connecting to a Time Tagger over a network: *Network Time Tagger* and *Pyro5*.

Network Time Tagger is an ideal solution for sharing a Time Tagger between different computers. By using *Network Time Tagger*, a remote computer has direct access to the Time Tag stream and can perform measurements locally, as if the Time Tagger was directly connected over USB. *Pyro* on the other hand can be used to access a Time Tagger remotely. From a remote computer, *Pyro* can start measurements on the computer connected to the time tagger and return the results.

3.4.1 Sharing a Time Tagger with Network Time Tagger

From Time Tagger software version 2.10 onward, Time Tagger supports remote operation ‘out-of-the-box’ with *Network Time Tagger*. *Network Time Tagger* implements a server on a computer connected to a Time Tagger and sends the Time Tag stream directly to the clients. Clients can then connect to the server and run arbitrary measurements independently on their own computers as if they are directly connected to the hardware device.

The server can be set up with the Time Tagger Graphical User Interfaces (Time Tagger Lab or Web Application) or the Time Tagger API through `TimeTagger.startServer()`. When setting up the server, the host can decide on the level of access for the clients. With `AccessMode.Control()`, the clients have full access to data from all channels and can change the settings of the Time Tagger. With `AccessMode.Listen()`, the host can decide to only share data from specific channels.

Clients can connect to the server using the Time Tagger API with `createTimeTaggerNetwork()`. Once connected, measurements can be performed directly on the client side. Function calls are identical to the ones used to control a Time Tagger locally. Therefore, programs written with the Time Tagger API can be easily adapted to run on a remote computer.

Note: Network Time Tagger can also be used to access a Time Tagger with different programming languages at the same time, both locally or on a remote computer. With `createTimeTaggerNetwork()`, it is for example possible to run simultaneous measurements with Matlab and Python.

Below is a minimal working example for setting up a Network Time Tagger server and client with Python 3.6.

Listing 1: Starting a Network Time Tagger server

```
import TimeTagger

tagger = TimeTagger.createTimeTagger()
#connect to the Time Tagger via USB

tagger.startServer(access_mode = TimeTagger.AccessMode.Control,port=41101)
# Start the Server. TimeTagger.AccessMode sets the access rights for clients. Port_
↪ defines the network port to be used
# The server keeps running until the command tagger.stopServer() is called or until the_
↪ program is terminated
```

Listing 2: Connecting to a Network Time Tagger server

```
import TimeTagger

tagger = TimeTagger.createTimeTaggerNetwork('ip:port')
# Connect to the Time Tagger server. 'ip' is the IP address of the server and 'port' is the_
↪ port defined by the server. The default port is 41101

correlation = TimeTagger.Correlation(tagger=tagger, channel_1=1, channel_2=2, binwidth=1,
↪ n_bins=1000)
# tagger can be used to perform measurements as if the client was connected to the_
↪ TimeTagger via USB. In this case, the client starts a correlation measurement.
# After a measurement is finished, the client can disconnect with TimeTagger.
↪ freeTimeTagger(tagger)
```

3.4.2 Remote control of a Time Tagger with Pyro

Pyro5 is a Python library that allows operation of a Time Tagger from a remote computer. It is able to send API commands to the remote Time Tagger and to obtain their return values. In the following, we describe how to use Pyro5 and achieve seamless access to the *Time Tagger's API* remotely.

Listing 3: Teaser code

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

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

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

(continues on next page)

(continued from previous page)

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

3.4.3 Remote procedure call

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

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

3.4.4 Initial setup

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

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

The last missing part, the Pyro5 package, you can install from [PyPi](#) as

```
pip install Pyro5
```

3.4.5 Minimal example

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

Server code

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

```
import Pyro5.api
import TimeTagger as TT

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

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

(continues on next page)

(continued from previous page)

```

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

```

Client code

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

```

import Pyro5.api

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

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

>> ['17400000ABC', '17500000ABC']

```

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

3.4.6 Creating the Time Tagger

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

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

```

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

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

    def createTimeTagger(self):

```

(continues on next page)

(continued from previous page)

```

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

```

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

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

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

```

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

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

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

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

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

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

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

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

    def freeTimeTagger(self, tagger_proxy):
        """Free Time Tagger. """
        # Client only has a proxy object.

```

(continues on next page)

(continued from previous page)

```

objectId = tagger_proxy._pyroUri.object
# Get adapter object from the server.
tagger = self._pyroDaemon.objectsById.get(objectId)
self._pyroDaemon.unregister(tagger)
return TT.freeTimeTagger(tagger._obj)

```

3.4.7 Measurements and virtual channels

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

```

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

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

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

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

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

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

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

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

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

@Pyro5.api.expose
class DelayedChannel():
    """Adapter class for DelayedChannel."""

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

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

```

(continues on next page)

(continued from previous page)

```

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

    # Earlier code omitted (...)

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

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

```

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

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

3.4.8 Working example

Download the complete source files

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

Start the server in a terminal window:

```
> python simple_server.py
```

Now open a second terminal window and run the example:

```
> python simple_example.py
```

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

Listing 4: simple_example.py

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

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

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

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

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

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

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

See also:

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

3.4.9 What is next?

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

[Let us know](#) if you have any questions about RPC interface for the Time Tagger.

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

SYNCHRONIZER

4.1 Overview

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

4.2 Key applications

The Synchronizer offers numerous advantages beyond its capability to extend the number of input channels up to 144, when connected to 8 Time Taggers, without introducing any additional time jitter.

4.2.1 Crosstalk elimination

Employing synchronized Time Taggers provides the benefit of effectively suppressing analog crosstalk in comparison to measurements involving a single device unit. In experiments such as interferometry, for instance, the crosstalk pickup in correlation measurements hinders the identification of the physical signal from the electronic noise around zero. A commonly adopted solution consists on delaying signals with respect to each other by using cables of different lengths. Nevertheless, this approach might introduce additional complexities, e.g. the length difference fluctuates over time due to temperature variations. In this regard, the Synchronizer represents a more robust solution.

4.2.2 High transfer rate

The data transfer rate from a single Time Tagger to the PC is limited to 80 MTags/s by the single-core CPU performance. Utilizing more than one Time Tagger allows to use multiple USB controllers and CPU threads, significantly increasing the total transfer rate (up to 80 MTags/s per device), as far as the Time Tagger and PC processing capabilities are not overcome.

4.2.3 Multi-room experiment

In some experiments, Time Taggers need to be placed far apart. The cable length between the Synchronizer and the Time Tagger can be as long as 50 m, without any additional time jitter caused. Different Sync cables result only in a constant time offset between the signals.

4.2.4 Synchronizer with a single Time Tagger

You can benefit from the Synchronizer with a single Time Tagger at least in the following two application scenarios:

1. **Long term clock stability**

The Synchronizer contains a highly stable clock oscillator which you can benefit from even when you have only one Time Tagger. Just connect any clock output of the Synchronizer to the *CLK IN* input of the Time Tagger and enjoy the clock stability provided by the Synchronizer, which matters especially measuring long time differences.

2. **Absolute clock timestamps**

By connecting all signals from the Synchronizer as shown in [Cable connections](#) the timestamps in the Time Tag stream will be referenced to the power-up time of the Synchronizer. Even when you disconnect the Time Tagger from your PC, e.g., in case of power down, USB timeout, or software restart, the time tags returned by the Time Tagger will remain referenced to the start time of the Synchronizer. To verify that this configuration is active, you will see a warning message in the console on `createTimeTagger()` that you are using the Synchronizer with only one Time Tagger.

4.3 Requirements

Successful synchronization of your Time Taggers requires:

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

4.4 Cable connections

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

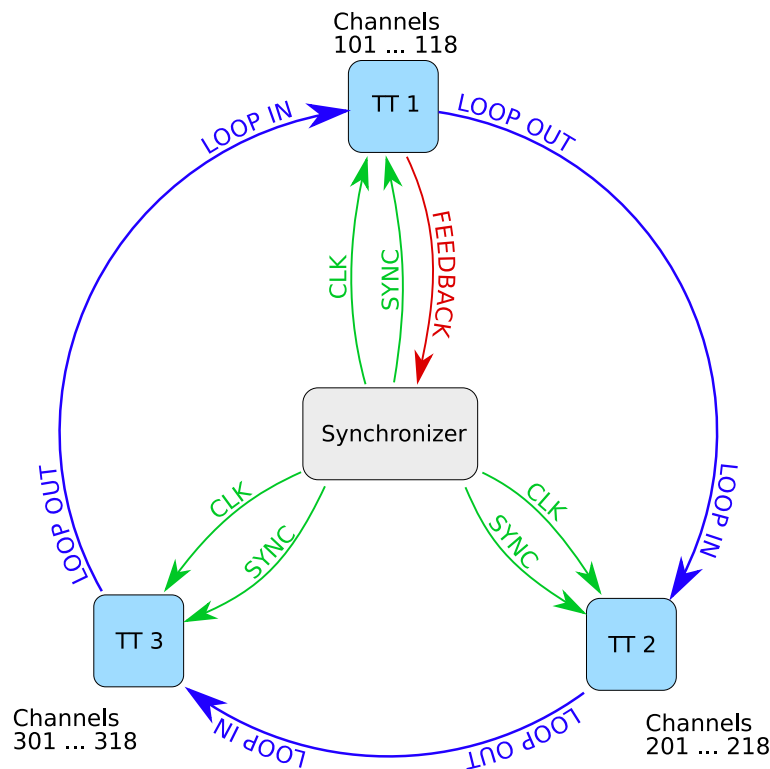
Using a single Time Tagger with Synchronizer is also possible and you shall connect *LOOP IN* and *LOOP OUT* together on the same device.

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

Table 1: Connections between the Synchronizer and Time Taggers

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

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



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

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

USB port simultaneously. Typically, such USB controllers have an individual chip for each USB port and require a PCIe x4 slot on the computer's motherboard..

4.4.1 Using an external reference clock

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

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

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

Table 3: Signal parameters at *REF OUT*.

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

4.5 Software and channel numbering

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

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

```
tagger = createTimeTagger()
```

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

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

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

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

Channel Number Schema 0 and 1.

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

```
from TimeTagger import createTimeTagger, TT_CHANNEL_RISING_EDGES

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

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

4.5.1 Incomplete cable connections

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

4.5.2 Buffer overflows

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

4.6 Limitations

4.6.1 Conditional filter

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

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

4.6.2 Internal test signal

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

4.7 Status LEDs and troubleshooting

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

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

The LEDs of the *Time Tagger Ultra* also indicate the state of the synchronization loop. See more details in section [LEDs](#).

HARDWARE

5.1 Input channels

The Time Tagger has 8 or 18 inputs (SMA-connectors). The electrical characteristics are tabulated below. Each input can detect both, rising and falling edges of an input pulse, and each input has two channels associated with it. Rising edges correspond to channel numbers 1 to 18 (*Time Tagger 20*: 1 to 8) and falling edges correspond to respective channel numbers -1 to -18 (*Time Tagger 20*: -1 to -8). Thereby, you can treat rising and falling edges in a fully equivalent fashion.

5.1.1 Electrical characteristics

Property	<i>Time Tagger 20</i>	<i>Time Tagger Ultra</i>	<i>Time Tagger X</i>
Termination	50 Ohm	50 Ohm	50 Ohm / High-Z
Input voltage range (recommended)	0.0 to 3.0 V	-3.0 to 3.0 V	-1.5 to 1.5 V
Maximum input (no damage)	-0.3 to 5.0 V	-5.0 to 5.0 V	-3.0 to 3.0 V
Trigger level range	0.0 to 2.5 V	-2.5 to 2.5 V	-1 to 1 V
Minimum signal level	100 mV	100 mV	100 mV
Minimum pulse width	1.0 ns	0.5 ns	350 ps

5.1.2 Configurable input termination - Time Tagger X only

The input termination of the *Time Tagger X* is configurable during runtime to either 50 Ohm or High-Z (see [*TimeTagger.setInputImpedanceHigh\(\)*](#)). Usually 50 Ohm should be chosen to accomplish proper HF termination, but High-Z is useful in certain use cases with small amplitudes and weak output drivers.

Caution: When the *Time Tagger X* is unpowered or not configured (before [*createTimeTagger\(\)*](#) has been called), the input termination is High-Z. This is to protect the *Time Tagger's* input stage from potentially damaging operating conditions (e.g. signals into an unpowered input stage). Since software version 2.17.0, the termination does not switch switch to 50 Ohm upon initialization anymore. However, the channel will switch to 50 Ohm by default as soon as it is registered. To prohibit this switching behavior, set the impedance explicitly to High-Z by [*TimeTagger.setInputImpedanceHigh\(\)*](#) before the first usage of the respective input, e.g. in a measurement.

One of the following measures can be taken when connecting signal sources to the *Time Tagger X* which are sensitive to operation without termination:

- The signal source is only operated after the *Time Tagger X* is powered and configured properly. The *Time Tagger's* input termination is set to 50 Ohm.

- An external 50 Ohm termination is connected between SMA cable and the *Time Tagger's* input port. The *Time Tagger's* input termination is set to High-Z.
- An HF circulator or isolator is connected to the output of the signal source to prevent any potentially damaging reflections from getting into the output.

5.1.3 High Resolution Mode

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

The tables show the usable inputs for the different modes. Channels available with the minimal four-channel license are shown without parenthesis. When additional channels are added, the priority will be given to the HighRes ones.

Table 1: Time Tagger Ultra Performance

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

Table 2: Time Tagger X

Mode	HighRes channels	Standard channels
Standard		1 - 4, (5 - 18)
HighResB	1, 5, 9, 13, (17)	

Note: As a result of the averaging process, the quality of the calculated timestamps is affected by relative changes in the internal delays of the contributing inputs. These delays are especially affected by the device's temperature. It is strongly recommended to let the device heat up for at least 10 s before starting a measurement. Constant average count rates (averaged over the timescale of hundreds of milliseconds) will provide the best results. If you need more information on this topic, please contact us via support@swabianinstruments.com.

5.2 Data connection

The *Time Tagger 20* is powered via a USB connection. Therefore, you should ensure that the USB port is capable of providing the full specified current (500 mA). A USB \geq 2.0 data connection is required for the performance specified here. Operating the device via a USB hub is strongly discouraged. The *Time Tagger 20* can stream about 9 MTags/s.

The *Time Tagger Ultra* and *Time Tagger X* has a USB 3.0 interface. This allows to stream up to 80 MTags/s to the PC. The actual number highly depends on the performance of the CPU the Time Tagger is connected to and the evaluation methods involved.

In addition, the *Time Tagger X* is equipped with both an SFP+ Port (10 GbE) and a QSFP+ port (40 GbE) which can be used for streaming up to 300 MTags/s or 1200 MTags/s respectively.

5.3 LEDs

The Time Tagger devices have LEDs showing status information.

5.3.1 Time Tagger X

Front panel and power button

On its front panel, the *Time Tagger X* has an LED inside the power button and individual channel status LEDs:

Table 3: Power button LED

Color	Description
blue	Device in standby, press button to turn it on
green	Device running
orange	Device is getting ready
red	An error occurred

Table 4: Channel LEDs

Color	Description
dark	Channel unavailable (according to your license)
blue	Channel available but not used by a measurement
solid green	Measurement running but no data within last 2 s
blinking green	Time tags are streamed to the PC. Blinking frequency indicates data rate
solid orange	Overflow
solid red	Error

Rear panel

Table 5: LED next to the *CLK IN* input

Color	Description
dark	No clock signal
solid green	Valid reference or synchronization clock
solid red	Invalid reference frequency
solid blue	Ext. clock valid, but not in use

Table 6: LED next to the *SYNC IN* input

Color	Description
dark	No synchronizer on <i>CLK</i> input
green	Valid signal at <i>SYNC IN</i>
red	Invalid signal at <i>SYNC IN</i>

Table 7: LED next to the *LOOP IN* input

Color	Description
dark	No synchronizer on <i>CLK</i> input
green	Valid signal at <i>LOOP IN</i>
red	Invalid signal at <i>LOOP IN</i>

5.3.2 Time Tagger Ultra

The “Power” LED turns green when the power is supplied to the device.

Table 8: Status LED

Color	Description
solid green	Firmware loaded
blinking green	Time tags are streaming
solid orange	Overflows occurred. LED turns orange for 0.3 s on overflow events. Solid orange indicates continuous overflows.
solid red	Device initialization failed (check USB connection)

Table 9: LED next to the *CLK* input

Color	Description
dark	No clock signal
solid green	Valid reference or synchronization clock
solid red	Invalid reference frequency
solid blue	Ext. clock valid, but not in use
fast blinking red	Calibration error on at least one channel
blinking red (hardware <v1.5)	Invalid signal at <i>SYNC IN</i> (<i>AUX IN 1</i>)
blinking yellow (hardware <v1.5)	Invalid signal at <i>LOOP IN</i> (<i>AUX IN 2</i>)

Table 10: LED next to the *SYNC IN* input (hardware \geq v1.5)

Color	Description
dark	No synchronizer on <i>CLK</i> input
green	Valid signal at <i>SYNC IN</i>
red	Invalid signal at <i>SYNC IN</i>

Table 11: LED next to the *LOOP IN* input (hardware $\geq v1.5$)

Color	Description
dark	No synchronizer on <i>CLK</i> input
green	Valid signal at <i>LOOP IN</i>
red	Invalid signal at <i>LOOP IN</i>

5.3.3 Time Tagger 20

The “Power” LED turns green when the power is supplied to the device.

Table 12: Status LED

Color	Description
solid green	Firmware loaded
blinking green-orange	Time tags are streaming
red	Overflows occurred. LED turns red for 0.1 s on every overflow event. Solid red indicates continuous overflows.
solid blue	Device initialization failed (check USB connection)

5.4 Test signal

The Time Tagger has a built-in test signal generator that generates a square wave with a frequency in the range 0.8 to 1.0 MHz. You can apply the test signal to any input channel instead of an external input. This is especially useful for testing, calibrating and setting up the Time Tagger initially. The *Time Tagger X* also provides the opportunity to put out two square wave signals with a variable frequency via the AUX Out ports on the back of the device.

5.5 Synthetic input delay

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

5.6 Synthetic dead time

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

5.7 Event divider

You can introduce an event divider for each channel independently. This is useful to discard a given number of time tags before the next one is stored, e.g., to reduce the data transfer rate requirement at expense of the data accumulation efficiency. The event divider can be set individually for rising and for falling edges.

5.8 Conditional Filter

The Conditional Filter allows you to decrease the time tag rate without losing those time tags that are relevant to your application, for instance, where you have a high-frequency signal applied to at least one channel. Examples include fluorescence lifetime measurements or optical quantum information and cryptography, where you want to capture synchronization clicks from a high repetition rate excitation laser.

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

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

Note: In *Time Tagger 20*, the software-defined input delays, as set by the method `setInputDelay()`, do not apply to the Conditional Filter logic.

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

5.9 Bin equilibration

The discretization of electrical signals is never perfect. In time-to-digital conversion, this manifests as small differences (few ps) in 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 in the time resolution by $\sqrt{2}$. This feature is enabled by default. If your application requires time resolution down to the jitter limit, you can disable this feature.

5.10 Overflows

The *Time Tagger 20* is capable of continuous streaming of about 9 MTags/s. For the *Time Tagger Ultra* and *Time Tagger X*, continuous tags streamed can exceed 80 MTags/s, depending on the CPU of the PC the Time Tagger is attached to. Higher data rates for short times are buffered internally, so that no overflow occurs. If continuous higher data rates persist, the internal buffer gets completely filled. Therefore, some of the time tags are discarded and not transferred to the PC, resulting in data loss. The hardware allows you to check with `TimeTaggerBase.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 data blocks and discard the invalid tags in between. Your measurement data may still be valid, although your acquisition time will likely increase.

5.11 External Clock Input

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

Time Tagger X and Time Tagger Ultra

The external clock input can be used to synchronize different Time Tagger devices. The input clock frequency must be 10 or 500 MHz. The *CLK* input requires between 100 mVpp and 4 Vpp AC coupled into 50 Ohm, 500 mVpp is recommended. The lock status can be read off the LED color: If the *CLK* LED shines green, the *Time Tagger* is locked and uses the provided clock. If the LED is blue, a valid frequency is supplied, however, the Time Tagger is still configured to use the internal clocking source. In case of a wrong or unstable frequency, the LED will shine red. A 500 MHz *CLK* input without a Synchronizer will lead to red LEDs on *LOOP IN* and *SYNC IN*. Yet, the Time Tagger will still work normally and the LEDs can be ignored in this case.

External clock signal requirements:

The input clock signal must have a very low jitter to provide the specified performance of the *Time Tagger*. Please note that the timing specifications for the *Time Tagger Ultra* with respect to other devices on the same clock are only met from hardware version 2.3 and later.

Caution: In order to reach the specified input jitter for the Time Tagger with an external clock, the input signals must be uncorrelated to the external clock. This restriction does not exist for `TimeTaggerBase.setSoftwareClock()`.

Time Tagger 20

The *Time Tagger 20* supports software clock feature only.

5.12 Synchronization signals

Time Tagger X and Time Tagger Ultra

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

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

Time Tagger 20

Synchronization of multiple *Time Tagger 20* devices is not possible.

5.13 FPGA link

Time Tagger X

The Ethernet based FPGA link can be used for connecting customer's FPGA designs directly to the *Time Tagger X*. The connection is provided through either SFP+ or QSFP+ connector on the back panel of the *Time Tagger X*. Either one of them can be active and shall be used for connection of customer's design using either a Direct Attach Cable (DAC) or optical fiber transceiver. More details and explanations can be found in the [In Depth Guide: FPGA link](#).

Time Tagger Ultra and Time Tagger 20

Time Tagger Ultra and *Time Tagger 20* have no support for FPGA link.

5.14 General purpose IO (GPIO)

Time Tagger Ultra

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

Time Tagger 20

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

SOFTWARE OVERVIEW

At the heart of the *Time Tagger* software is a multi-threaded processing engine that receives the time tag stream and feeds it to all running measurements. The measurements and the virtual channels are parallel processing units that analyze the time tag stream each in their own way. For example, a count rate measurement analyzes all time tags from one or more specific channels and calculates the average number of tags received per second. A cross-correlation measurement compute the cross-correlation between two channels, typically by sorting the time tags in histograms, and so on. Such a powerful architecture enables you to perform any thinkable digital time domain measurement in real time. You have several choices on how to use this architecture.

6.1 Graphical User Interfaces

The easiest way of using the *Time Tagger* is one of the graphical user interfaces, either *Time Tagger Lab* (only on Windows OS) or the *Web Application*. They allow you to interact with the hardware on your computer or a tablet. You can create measurements, get live plots, and save and load the acquired data.

6.2 Precompiled libraries and high-level language bindings

We have implemented a set of commonly useful measurements including count rates, auto-correlation, cross-correlation, fluorescence lifetime imaging (FLIM (Fluorescence-lifetime imaging microscopy)), 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 this API 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 Programming Interface* if you plan to use the Time Tagger in this way.

6.3 C++ API

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

APPLICATION PROGRAMMING INTERFACE

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

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

7.1 Examples

Often the fastest way to get an impression on the API is through examples. Several examples for multiple programming languages are available in the Time Tagger installation folder.

7.1.1 Measuring cross-correlation

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

```
# Create an instance of the TimeTagger
tagger = createTimeTagger()

# Adjust trigger level on channel 2 to 0.25 Volt
tagger.setTriggerLevel(2, 0.25)

# Add time delay of 123 picoseconds on the channel 3
tagger.setInputDelay(3, 123)

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

(continues on next page)

(continued from previous page)

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

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

7.1.2 Using virtual channels

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

```
tagger = createTimeTagger()

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

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

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

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

>> [ 8000008.81  8000008.81 16000017.62]
```

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

7.1.3 Using multiple Time Taggers

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

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

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

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

7.1.4 Using Time Tagger remotely

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

You can start the server by calling `TimeTagger.startServer()` on an existing `TimeTagger` object.

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

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

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

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

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

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

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

# Connect to the server
ttn = createTimeTaggerNetwork(address)

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

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

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

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

7.2 The TimeTagger Library

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

7.2.1 Units of measurement

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

Analog voltage levels are specified in Volts.

7.2.2 Channel numbers

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

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

7.2.3 Unused channels

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

7.2.4 Constants

CHANNEL_UNUSED

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

7.2.5 Enumerations

class AccessMode

Controls how the Time Tagger server delivers the data-blocks to the connected clients, and if the clients are allowed to change the hardware settings.

Control

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

Listen

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

SynchronousControl

The same as [AccessMode.Control](#) but the data is delivered synchronously to every client.

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

class ChannelEdge

Selects the channels that [TimeTagger.getChannelList\(\)](#) returns.

All

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

Rising

Rising edges of channels with HighRes and Standard resolution.

Falling

Falling edges of channels with HighRes and Standard resolution.

HighResAll

Rising and falling of channels edges with HighRes resolution.

HighResRising

Rising edges of channels with HighRes resolution.

HighResFalling

Falling edges of channels with HighRes resolution.

StandardAll

Rising and falling edges of channels with Standard resolution.

StandardRising

Rising edges of channels with Standard resolution.

StandardFalling

Falling edges of channels with Standard resolution.

class CoincidenceTimestamp

Defines what timestamp to use for a coincidence event in [Coincidence/Coincidences](#).

Last

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

Average

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

First

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

ListedFirst

Use the timestamp of the channel at the first position of the list when [Coincidence](#) or a group of [Coincidences](#) is instantiated.

class FpgaLinkInterface

Determines which Ethernet Port on the *Time Tagger X* should be used in `TimeTagger.enableFpgaLink()`.

SFPP_10GE

Use the SFP+ Port on the *Time Tagger X* for FPGA link output.

QSFP_40GE

Use the QSFP+ Port on the *Time Tagger X* for FPGA link output.

class GatedChannelInitial

The initial state of a [GatedChannel](#).

Closed = 0

The gate is closed initially.

Open = 1

The gate is open initially.

class Resolution

Defines the resolution mode of the Time Tagger on connection using `createTimeTagger()`. Details on the available inputs are listed in the [hardware overview](#).

Standard

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

HighResA

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

HighResB

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

HighResC

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

class TagType

This enumeration describes the overflow condition.

TimeTag = 0

A normal event from any input channel, no overflow.

Error = 1

An error in the internal data processing, e.g. on plugging the external clock. This invalidates the global time.

OverflowBegin = 2

Marks the beginning of an interval with incomplete data because of too high data rates.

OverflowEnd = 3

Marks the end of the interval. All events, which were lost in this interval, have been handled

MissedEvents = 4

This virtual event signals the number of lost events per channel within an overflow interval. Might be sent repeatedly for larger number of lost events.

class UsageStatisticsStatus**Disabled = 0**

Usage statistics collection and upload is disabled.

Collecting = 1

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

CollectingAndUploading = 2

Enable usage statistics collection and automatic upload

7.2.6 Functions

createTimeTagger([serial=", resolution=Resolution.Standard])

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

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

In MATLAB, this function is accessed as `TimeTagger.createTimeTagger`.

Parameters

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

Returns

TimeTagger object

Return type

TimeTagger

Raises

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

createTimeTaggerVirtual()

Creates a virtual Time Tagger object. Virtual Time Tagger uses time-tag dump file(s) as a data source instead of Time tagger hardware. This allows you to use all Time Tagger library measurements for offline processing of the dumped time tag stream. For example, you can repeat the analysis of your experiment with different parameters, like different binwidths etc.

In MATLAB, this function is accessed as `TimeTagger.createTimeTaggerVirtual`.

Returns

TimeTaggerVirtual object

Return type

TimeTaggerVirtual

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

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

In MATLAB, this function is accessed as `TimeTagger.createTimeTaggerNetwork`.

Parameters

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

Returns

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

Return type

TimeTaggerNetwork

Raises

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

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

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

Parameters

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

Returns

Information about server, available channels and exposed channels.

Return type

str or *dict*

Raises

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

freeTimeTagger(*tagger*)

Releases all Time Tagger resources and terminates the active connection.

Parameters

tagger (*TimeTaggerBase*) – TimeTaggerBase object to disconnect

scanTimeTagger()

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

In MATLAB this function is accessible as `TimeTagger.scanTimeTagger()`.

Returns

List of serial numbers

Return type

list[str]

scanTimeTaggerServers()

Scans the network for available Time Tagger servers.

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

Returns

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

Return type

`list[str]`

setLogger(callback)

Registers a callback function, e.g. for customized error handling. Please see the examples in the installation folder on how to use it. Callback function shall have the following signature `callback(level, message)`. By default, the log messages are printed into the console.

Python example:

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

Matlab example:

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

setTimeTaggerChannelNumberScheme(int scheme)

Deprecated since version 2.17: all values of *scheme* except for 1 are deprecated

Selects whether the first physical channel starts with 0 or 1.

This method is deprecated and will be removed soon. The only purpose of this method is to call `setTimeTaggerChannelNumberScheme(TT_CHANNEL_NUMBER_SCHEME_ONE)` (with `TT_CHANNEL_NUMBER_SCHEME_ONE = 2`) before `createTimeTagger()` for old devices (channel numbers starting with 0) which will suppress a deprecation warning.

Important: The method must be called before the first call to `createTimeTagger()`.

getTimeTaggerChannelNumberScheme()

Deprecated since version 2.17.

Returns the currently used channel scheme.

Returns

Channel scheme

Return type

`int`

mergeStreamFiles(*output_filename*, *input_filenames*, *channel_offsets*, *time_offsets*, *overlap_only*)

Parameters

- **output_filename** (*str*) – Filename where to store the merge result *.ttbin.
- **input_filenames** (*List[str]*) – List of dump files that will be merged
- **channel_offsets** (*List[int]*) – Channel number offset for each *.ttbin file. Useful when input files have the same channel numbers.
- **time_offsets** (*List[int]*) – Time offset for each *.ttbin file in picoseconds.
- **overlap_only** (*bool*) – If True, then merge only the regions where the time is overlapping.

This function merges a list of time tag stream files into one file. The merged stream file can be loaded into the [TimeTaggerVirtual](#) for processing. The file merging combines streams into one with the possibility of specifying a constant time offset for each input stream file. Additionally, it is possible to specify channel number offset if the input stream files were recorded from the same channel numbers, for instance, using two Time Tagger devices. The parameters *input_filenames*, *channel_offsets*, and *time_offsets* shall be of equal length.

This function handles the *.ttbin files the same way as the [TimeTaggerVirtual.replay\(\)](#).

See also: [FileWriter](#), [FileReader](#), and [The TimeTaggerVirtual class](#).

Note: When merging multiple stream files recorded at different times or from different devices, you have to be aware of possible time base differences. This function does not rescale the data into a common time base as this would require additional information and external synchronization signal. If you want to improve the synchronicity of the time base between two devices, please send the reference clock signal to any of the available inputs of each Time Tagger and set up the software clock [setSoftwareClock\(\)](#).

getVersion()

Returns

Version of the Time Tagger software.

Return type

str

Usage statistics data collection

See also the section [Usage Statistics Collection](#).

setUsageStatisticsStatus(*status*)

Parameters

status ([UsageStatisticsStatus](#)) – New status of the usage statistics data collection.

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

getUsageStatisticsStatus()

Returns

Returns the current status of the usage statistics for the current user. The status is described by the [UsageStatisticsStatus](#).

Return type*UsageStatisticsStatus***getUsageStatisticsReport()**

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

Returns

Usage statistics data encoded as JSON string.

Return type*str*

7.2.7 Helper classes

```
class ChannelGate(gate_open_channel: int, gate_close_channel: int, initial: GatedChannelInitial =
                  GatedChannelInitial.Open)
```

Parameters

- **gate_open_channel** (*int*) – Number of the channel that opens the evaluation gate.
- **gate_close_channel** (*int*) – Number of the channel that closes the evaluation gate.
- **initial** (*GatedChannelInitial*) – Initial state of the evaluation gate.

This object defines an evaluation gate that is passed to a measurement class. The time-tag stream itself is not modified but sections of the stream can be excluded from the evaluation. In contrast to time-tag stream based gating (see *GatedChannel*), this concept allows the measurement class to calculate the correct data normalization.

7.3 TimeTagger classes

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

Time Tagger hardware

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

Virtual Time Tagger

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

Network Time Tagger

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

7.3.1 General Time Tagger features

The *TimeTaggerBase* class defines methods and functionality present in all Time Tagger objects. The specific classes below inherit from *TimeTaggerBase*. Every *measurement* and *virtual channel* instance requires a reference to a *TimeTaggerBase* object to associate with.

class TimeTaggerBase

setInputDelay(*channel*, *delay*)

Convenience method that calls *setDelaySoftware()* if you use a *Time Tagger 20* or the delay is > 2 μ s, otherwise *setDelayHardware()* is called.

Parameters

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

getInputDelay(*channel*)

Convenience method that returns the sum of *getDelaySoftware()* and *getDelayHardware()*.

Parameters

channel (*int*) – Channel number

Returns

Delay time in picoseconds

Return type

int

setDelayHardware(*channel*, *delay*)

Note: Method is not available for the *Time Tagger 20*.

Set an artificial delay per *channel*. The delay can be positive or negative. This delay is applied onboard the Time Tagger directly after the time-to-digital conversion, so it also affects the *Conditional Filter*. If you exceed the maximum hardware delay range, please use *setDelaySoftware()* instead.

Parameters

- **channel** (*int*) – Channel number
- **delay** (*int*) – Delay time in picoseconds, the maximum/minimum value allowed is ± 20000000 (± 2 μ s)

getDelayHardware(*channel*)

Note: Method is not available for the *Time Tagger 20*.

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

Parameters

channel (*int*) – Channel number

Returns

Delay time in picoseconds

Return type

int

setDelaySoftware(*channel*, *delay*)

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

Parameters

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

getDelaySoftware(*channel*)

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

Parameters

channel (*int*) – Channel number

Returns

Delay time in picoseconds

Return type

int

setDeadtime(*channel*, *deadtime*)

Sets the dead time of a channel in picoseconds. The minimum dead time is defined by the internal clock period, which is 6 ns for the *Time Tagger 20*, 2 ns for the *Time Tagger Ultra*, and 1.333 ns for the *Time Tagger X*. For the *Time Tagger 20*, the requested dead time will be rounded to the nearest multiple of the 6 ns clock cycle. The other models allow for arbitrary dead times greater than the respective minimum dead time.

As the dead time passed as an input might be altered to the rounded value, the rounded value will be returned. The maximum dead time is 393 μ s for the *Time Tagger 20*, 2147 μ s for the *Time Tagger Ultra*, and 716 μ s for the *Time Tagger X*. Larger dead times will result in an exception.

Note: The specified dead time is 2.1 ns for *Time Tagger Ultra* and 1.5 ns for *Time Tagger X*. With the default setting of the hardware dead time filter, an event arriving between the default hardware dead time and the specified dead time after the last event of that channel might be dropped (e.g., an event arriving between 2 ns and 2.1 ns after the last event on that channel for *Time Tagger Ultra*).

Parameters

- **channel** (*int*) – Channel number
- **deadtime** (*int*) – Dead time value in picoseconds

Returns

Resulting dead time in picoseconds, that might be rounded to the nearest valid value (minimum dead time or multiple of the clock period).

Return type

int

getDeadtime(*channel*)

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

Parameters

channel (*int*) – Physical channel number

Returns

Dead time value in picoseconds

Return type`int`**getOverflows()**

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

Returns

Number of overflows

Return type`int`**getOverflowsAndClear()**

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

Returns

Number of overflows

Return type`int`**clearOverflows()**

Set the overflow counter to zero.

setSoftwareClock(*input_channel*: `int`, *input_frequency*: `float`, *averaging_periods*: `float` = 1000, *wait_until_locked*: `bool` = True)

Define in software one of the input channels as the base clock for all channels. This feature sets up a software PLL (Phase-locked loop) and rescales all incoming time-tags according to the software clock defined. The PLL provides a new time base with “ideal clock tags” separated by exactly the defined *clock_period*. For measurements, you can use both, rescaled and ideal clock tags.

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

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

For the *Time Tagger 20*, a phase error of 200 ps needs to be considered when using the software clock.

Parameters

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

- **wait_until_locked** (*bool*) – Blocks the execution until the software clock is locked. Throws an exception on locking errors. All locking log messages are filtered while this call is executed. Default: True

disableSoftwareClock()

Disable the software clock.

getSoftwareClockState()

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

Returns

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

Return type

SoftwareClockState

getFence(*alloc_fence: bool = True*)

Generate a new fence object, which validates the current configuration and the current time. This fence is uploaded to the earliest pipeline stage of the Time Tagger. Waiting on this fence ensures that all hardware settings, such as trigger levels, channel registrations, etc., have propagated to the FPGA and are physically active. Synchronizes the Time Tagger internal memory so that all tags arriving after the *waitForFence()* call were actually produced after the *getFence()* call. The *waitForFence()* function waits until all tags, which are present at the time of the function call within the internal memory of the Time Tagger, are processed. This call might block to limit the number of active fences.

Parameters

alloc_fence (*bool*) – optional, default: True. If False, a reference to the most recently created fence will be returned instead

Returns

The allocated fence

Return type

int

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

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

Parameters

- **fence** (*int*) – fence object, which shall be waited on
- **timeout** (*int*) – (optional) Timeout in milliseconds. Negative means no timeout, zero returns immediately. Default: -1.

Returns

True if the fence has passed, false on timeout

Return type

bool

sync(*timeout: int = -1*)

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

The operation of this method on the *TimeTaggerNetwork* depends on the server access mode. If the *TimeTaggerNetwork* is connected to the Time Tagger server started in *AccessMode.Control*, the synchronization will be done all way through the server and the hardware. If the Time Tagger server started in *AccessMode.Listen*, the client will be able to synchronize only with the server but will not synchronize with the Time Tagger Hardware. However, if a USB synchronization fence was created by the server side, the clients will also see it.

See also:

- *getFence(), waitForFence(), startServer(), AccessMode*
- *Synchronization of the Time Tagger pipeline*

Parameters

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

Returns

True if the synchronization was successful, false on timeout

Return type

bool

getInvertedChannel(channel)

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

Parameters

channel (*int*) – Channel number

Returns

Channel number

Return type

int

isUnusedChannel(channel)

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

Parameters

channel (*int*) – Channel number

Returns

True/False

Return type

bool

getConfiguration()

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

Returns

Time Tagger settings and currently existing measurements.

Return type

str or *dict*

7.3.2 Time Tagger hardware

class TimeTagger

Base class: *TimeTaggerBase*

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

reset()

Reset the Time Tagger to the start-up state.

setTriggerLevel(channel, voltage)

Set the trigger level of an input channel in Volts.

Parameters

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

getTriggerLevel(channel)

Returns trigger level for the specified physical channel number.

Parameters

channel (*int*) – Physical channel number

Returns

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

Return type

float

getHardwareDelayCompensation(channel)

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

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

Parameters

channel (*int*) – Channel number

Returns

Hardware delay compensation in picoseconds

Return type

int

setConditionalFilter(trigger, filtered, hardwareDelayCompensation=True)

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

Parameters

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

- **hardwareDelayCompensation** (*bool*) – optional, default: True. If set to False, the physical hardware delay will not be compensated. This is only relevant for devices without `setDelayHardware()`, do not set this value to False if your device is capable of onboard delay compensation. Without onboard delay compensation, setting the value to False guarantees that the trigger tag of the conditional filter is always in before the triggered tag when the `InputDelays` are set to 0.

clearConditionalFilter()

Deactivates the event filter. Equivalent to `setConditionalFilter([], [], True)`. Enables the physical hardware delay compensation again if it was deactivated by `setConditionalFilter()`.

getConditionalFilterTrigger()

Returns the collection of trigger channels for the conditional filter.

Returns

List of channel numbers

Return type

`list[int]`

getConditionalFilterFiltered()

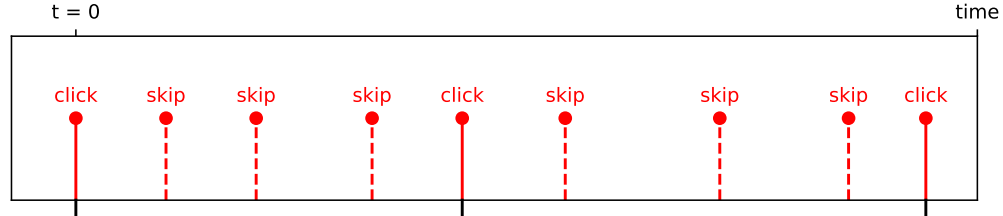
Returns the collection of channels to which the conditional filter is currently applied.

Returns

List of channel numbers

Return type

`list[int]`

setEventDivider(channel, divider)

Applies an event divider filter with the specified factor to a channel, which reduces the data rate. Only every *n*-th event from the input stream passes through the filter, as shown in the image. The divider is a 16 bit integer, so the maximum value is 65535.

Note that if the conditional filter is also active, the conditional filter is applied first.

Parameters

- **channel** (*int*) – Physical channel number
- **divider** (*int*) – Divider factor, min. 1 and max. 65535

getEventDivider(channel)

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

Parameters

channel (*int*) – Channel number

Returns

Divider factor value

Return type

`int`

setInputImpedanceHigh(*channel*, *state*)

Note: Method is only available for the *Time Tagger X*.

Sets the input impedance to High-Z for the specified *channel*. Before `createTimeTagger()`, after `TimeTagger.reset()`, and after `freeTimeTagger()`, the input is in the High-Z state. If not set explicitly to High-Z by `setInputImpedanceHigh()`, the input will switch to 50 Ohm by default as soon as the input is used.

Parameters

- **channel** (*int*) – Channel number
- **state** (*bool*) – True/False

```
# Upon initialization, all inputs are in the High-Z state:
tagger = TimeTagger.createTimeTagger()

# If you want to keep a channel in High-Z, set it right after initialization:
tagger.setInputImpedanceHigh(1, True)

# The Time Tagger will now stay in High-Z on channel 1, channel 2 will switch
→ to 50 Ohm:
cr = TimeTagger.CountRate(tagger, [1, 2])
```

getInputImpedanceHigh(*channel*)

Note: Method is only available for the *Time Tagger X*.

Returns whether the input impedance is set to high-Z for the specified *channel*.

Parameters

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

Returns

state of high input impedance

Return type

bool

setInputHysteresis(*channel*, *value*)

Note: Method is only available for the *Time Tagger X*.

Sets the input hysteresis value for the specified *channel*. Oscillations of the measured signal within the hysteresis range around the trigger value are ignored and therefore do not trigger new events. Supported values are 1 mV, 20 mV, 70 mV. Default input hysteresis value is 20 mV.

Parameters

- **channel** (*int*) – Channel number
- **value** (*int*) – hysteresis voltage value in mV (1, 20, 70)

getInputHysteresis(channel)

Note: Method is only available for the *Time Tagger X*.

Returns the voltage value in mV of the input hysteresis for the specified *channel*.

Parameters

channel (*int*) – Channel number

Returns

hysteresis voltage value in mV

Return type

int

setNormalization(channels, state)

Enables or disables Gaussian normalization of the detection jitter. Enabled by default.

Parameters

- **channels** (*list[int]*) – List of physical channel numbers
- **state** (*bool*) – True/False

getNormalization(channel)

Returns True if Gaussian normalization is enabled.

Returns

True/False

Return type

bool

setTestSignal(channels, state)

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

Parameters

- **channels** (*list[int]*) – List of physical channel numbers
- **state** (*bool*) – True/False

getTestSignal(channel)

Returns true if the internal test signal is activated on the specified *channel*.

Parameters

channel (*int*) – Physical channel number

Returns

True/False

Return type

bool

getSerial()

Returns the hardware serial number.

Returns

Serial number string

Return type

str

getModel()**Returns**

Model name as string

Return type

str

getPcbVersion()

Returns Time Tagger PCB (Printed circuit board) version.

Returns

PCB version

Return type

str

getDACRange()

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

Returns

Min and max voltage in Volt

Return type

(float, float)

getChannelList(*type=ChannelEdge.All*)Returns a list of channels corresponding to the given *type*.**Parameters****type** ([ChannelEdge](#)) – Limits the returned channels to the specified channel edge type**Returns**

List of channel numbers

Return type

list[int]

setHardwareBufferSize(*size*)Sets the maximum buffer size within the Time Tagger. The default value is 64 MTags, but can be changed within the range of 32 kTags to 512 MTags. Please note that this buffer can only be filled with a total data rate of up to 500 MTags/s. See also, *[Synchronization of the Time Tagger pipeline](#)*

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

Parameters**size** (*int*) – Buffer size, must be a positive number**autoCalibration()**

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

Returns

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

Return type

list[float]

getDistributionCount()

Returns the calibration data represented in counts.

Returns

Distribution data

Return type

2D_array[int]

getDistributionPSec()

Returns the calibration data in picoseconds.

Returns

Calibration data

Return type

2D_array[int]

getPsPerClock()

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

Returns

Clock period in picoseconds

Return type

int

setStreamBlockSize(max_events=131072, max_latency=20)

This option controls the latency and the block size of the data stream. Depending on which of the two parameters is exceeded first, the block stream size is adjusted accordingly.

Note: The block size will be reduced even further when no new tag arrives within roughly 1-2 μ s.

Parameters

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

setTimeTaggerNetworkStreamCompression(active)

Enables/disables the compression of TimeTags before they are streamed from the server to the clients. Activation can be helpful for slow network environments (≤ 100 MBit/s) if the bandwidth is the limiting factor. For instance, the amount of streamed data of periodic signals is reduced by about a factor of 2. The compression, on the other hand, leads to increased CPU utilization and is not advantageous for fast networks (≥ 1 GBit/s).

Parameters

active (bool) – flag defining whether the compression is enabled (default: False).

setTestSignalDivider(divider)

Change the frequency of the on-chip test signal.

For the *Time Tagger X*, the base frequency is 333.333 MHz and the default divider 375 corresponds to ~890 kCounts/s.

For the *Time Tagger Ultra*, the base frequency is 100.800 MHz and the default divider 126 corresponds to ~800 kCounts/s.

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

Parameters

divider (*int*) – Division factor

getTestSignalDivider()

Returns the value of test signal division factor.

getDeviceLicense()

Returns a JSON formatted string (*dict* in Python) containing license information of the Time Tagger device, for instance, model, edition, and available channels.

Returns

License information

Return type

dict

getSensorData()

Prints a JSON formatted string (*dict* in Python) containing all available sensor data for the given board. The *Time Tagger 20* has no onboard sensors.

Returns

Sensor data

Return type

dict

disableLEDs(*state*)

Disables all channel LEDs and back LEDs. The disabling of the power button LED will follow in the upcoming release.

Parameters

state (*bool*) – True/False

setLED(*bitmask*)

Manually change the state of the Time Tagger LEDs. The power LED of the *Time Tagger 20* cannot be programmed by software.

Example:

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

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

0 -> LED off

1 -> LED on

illumination bits

0-2: status, rgb - all Time Tagger models

3-5: power, rgb - *Time Tagger Ultra* only

6-8: clock, rgb - *Time Tagger Ultra* only

0 -> normal LED behavior, not overwritten by setLED

1 -> LED state is overwritten by the corresponding bit of 0-8

mask bits

16-18: status, rgb - all Time Tagger models

19-21: power, rgb - *Time Tagger Ultra* only

22-24: clock, rgb - *Time Tagger Ultra* only

Parameters

bitmask (*int*) – LED bitmask.

setSoundFrequency(*freq_hz*)

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

Parameters

freq_hz (*int*) – The sound frequency in Hz, use 0 to switch the buzzer off.

enableFpgaLink(*channels*, *destination_mac*, *link_interface* = *FpgaLinkInterface::SFPP_10GE*, *exclusive* = *False*)

Enable the FPGA link of the *Time Tagger X*

Parameters

- **channels** (*list[int]*) – List of channels, which shall be streamed over the FPGA link
- **destination_mac** (*str*) – Destination MAC, use an empty string for the broadcast address of "FF:FF:FF:FF:FF:FF"
- **link_interface** (*FpgaLinkInterface*) – selects which interface shall be used
- **exclusive** (*bool*) – Determines if time tags should exclusively be transmitted over Ethernet, increasing Ethernet performance and avoiding USB issues

disableFpgaLink()

Disable the FPGA link of the *Time Tagger X*

startServer(*access_mode*, *channels*=[], *port*=41101)

Start a Time Tagger server that can be accessed via [TimeTaggerNetwork](#). The server access mode controls if the clients are allowed to change the hardware parameters. See also: [AccessMode](#).

Parameters

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

Raises

[RuntimeError](#) – if server is already running.

stopServer()

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

isServerRunning()

Returns

True if server is running and False otherwise.

Return type`bool`

Note: The following *xtra* methods are mainly for development purposes and may be discontinued in future software versions without further notice. The *xtra* setter methods in this first section are only available for the *Time Tagger Ultra* and the *Time Tagger X*.

xtra_setAvgRisingFalling(channel, enable)

Configures if the rising and falling events shall be averaged. This is implemented on the device before any filter like event divider and it does not require to transfer both events. The primary application of this feature is the investigation of periodic signals with defined duty cycle.

They need to be manually delayed to be within a window of +-500 ps of error, else events might get lost. This condition restricts the applicable signals to pulses of constant duration. This method has no side effects on the channel `getInvertedChannel()`, you can still fetch the original events there. However if both are configured to return the averaged result, the timestamps will be identical.

Parameters

- **channel** (`int`) – The channel, on which the average value shall be returned
- **enable** (`bool`) – Select whether the averaging feature is enabled

xtra_getAvgRisingFalling(channel)

Return the state of the averaging of rising and falling edges.

Parameters

- **channel** (`int`) – The channel for which the averaging state is returned

Returns

The current enable state

Return type`bool`**xtra_setHighPrioChannel(channel, enable)**

Sets the priority state of a channel. This setting is applied on the hardware before USB transfer. If a buffer overflow occurs, channels with high-priority state will interrupt the overflow mode and be transmitted as standard time-tags (`TagType.TimeTag`). Timing information of low-priority channels is dismissed in overflow mode and only the number of counts is transmitted (`TagType.MissedEvents`). A typical application of the high-priority channels is `CountBetweenMarkers` with high-priority markers. In this case, the overflow range will be ideally sliced by the markers.

Parameters

- **channel** (`int`) – The channel on which the high-priority state shall be enabled
- **enable** (`bool`) – Select whether high priority is enabled

Caution: Interrupting the overflow mode may break the protection mechanism the overflow mode provides. This may lead to irreversible loss of events, not only loss of their timing information. High priority should only be assigned to low-count-rate channels, e.g. pixel triggers or similar control events.

xtra_getHighPrioChannel(channel)

Get the priority state of a channel.

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

The current enable state of the high-priority feature on this channel

Return type*bool*

Note: The following *xtra* methods are mainly for development purposes and may be discontinued in future software versions without further notice. The *xtra* setter methods in this second section are only available for the *Time Tagger X*.

xtra_setAuxOut(*channel*, *state*)Enables/Disables the Aux Out signal for the specified Aux *channel*.**Parameters**

- **channel** (*int*) – Aux channel number
- **state** (*bool*) – True/False

xtra_getAuxOut(*channel*)Returns whether the Aux Out signal is enabled for the specified Aux *channel*.**Returns**

State of the Aux Out signal

Return type*bool***xtra_setAuxOutSignal**(*channel*, *divider*, *duty_cycle*)Sets the signal shape, i.e., duty cycle and frequency, of the Aux out signal for the specified Aux *channel*.**Parameters**

- **channel** (*int*) – Aux channel number
- **divider** (*int*) – Divider of the Aux Out base signal frequency (333 MHz)
- **duty_cycle** (*float*) – The duty cycle of the aux signal

xtra_getAuxOutSignalDivider(*channel*):Returns the divider for the frequency of the Aux Out signal generator or the specified Aux *channel*.**Parameters****channel** (*int*) – Aux channel number**Returns**

Divider for the frequency of the Aux Out signal generator

Return type*int***xtra_getAuxOutSignalDutyCycle**(*channel*)Returns the duty cycle of the Aux Out signal for the specified Aux *channel*.**Parameters****channel** (*int*) – Aux channel number**Returns**

Duty cycle of the Aux Out signal generator

Return type

float

xtra_measureTriggerLevel(*channel*)

Measures and returns the applied voltage threshold of the specified *channel*.

Parameters

channel (*int*) – Channel number

Returns

Applied voltage threshold of a channel

Return type

float

xtra_setClockSource(*source*)

Specifies the different clock sources: 0 - internal clock , 1 - external clock 10 Mhz, 2 - external clock 500 MHz.

Parameters

source (*int*) – Number of the clock source. Allowed values: 0, 1, 2

xtra_getClockSource()

Returns the used clock source: 0 - internal clock , 1 - external clock 10 Mhz, 2 - external clock 500 MHz.

Returns

Number of the clock source

Return type

int

xtra_setClockAutoSelect(*state*)

Enables/Disables the auto clocking function.

Parameters

state (*bool*) – True/False

xtra_getClockAutoSelect()

Returns whether the auto clocking function is enabled.

Returns

State of auto clocking

Return type

bool

xtra_setClockOut(*state*)

Activates/Deactivates the 10 MHz clock output.

Parameters

state (*bool*) – True/False

7.3.3 The TimeTaggerVirtual class

In the Time Tagger software version 2.6.0, we have introduced the new *TimeTaggerVirtual*, which allows replaying earlier stored time-tag dump files. Using the virtual Time Tagger, you can repeat your experiment data analysis with different parameters or even perform different measurements.

Note: The virtual Time Tagger requires a free software license, which is automatically acquired from the Swabian Instruments license server when *createTimeTagger()* or *createTimeTaggerVirtual()* is called while a Time Tagger is attached. Once received, the license is permanently stored on this PC and the Virtual Time Tagger will work without Time Tagger hardware attached.

class TimeTaggerVirtual

Base class: *TimeTaggerBase*

replay(*file*, *begin*=0, *duration*=-1, *queue*=True)

Replay a dump file specified by its path *file* or add it to the replay queue. If the flag *queue* is false, the current queue will be discarded and file will be replayed immediately.

The *file* parameter can specify a header file or single specific file as shown in the following example.

```
# Assume we have following the files in the current directory:
# filename.ttbin
# filename.1.ttbin
# filename.2.ttbin

# Replay all files named "filename.NN.ttbin" sequentially
virtual_tagger.replay('filename.ttbin')

# Replay a single file "filename.1.ttbin"
virtual_tagger.replay('filename.1.ttbin')
```

See also: *FileWriter*, *FileReader*, and *mergeStreamFiles()*.

Parameters

- **file** (*str*) – the file to be replayed
- **begin** (*int*) – duration in picoseconds to skip at the beginning of the file. A negative time will generate a pause in the replay.
- **duration** (*int*) – duration in picoseconds to be read from the file. *duration*=-1 will replay everything. (default: -1)
- **queue** (*bool*) – flag if this file shall be queued. (default: *True*)

Returns

ID of the queued file

Return type

int

stop()

This method stops the current file and clears the replay queue.

waitForCompletion([*ID*=0, *timeout*=-1])

Blocks the current thread until the replay is completed.

This method blocks the current execution and waits until the given file has finished its replay. If no ID is provided, it waits until all queued files are replayed.

This function does not block on a zero timeout. Negative timeouts are interpreted as infinite timeouts.

Parameters

- **ID** (*int*) – selects which file to wait for. (default: 0)
- **timeout** (*int*) – timeout in milliseconds

Returns

true if the file is complete, false on timeout

Return type

bool

setReplaySpeed(*speed*)

Configures the speed factor for the virtual tagger.

A value of *speed*=1.0 will replay at a real-time rate. All *speed* values < 0.0 will replay the data as fast as possible but stops at the end of all data. This is the default value. Extreme slow replay speed between 0.0 and 0.1 is not supported.

Parameters

speed (*float*) – replay speed factor.

getReplaySpeed()

Returns the current speed factor.

Please see also [setReplaySpeed\(\)](#) for more details.

getConfiguration()

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

7.3.4 The TimeTaggerNetwork class

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

Note on performance

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

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

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

class TimeTaggerNetworkBase class: *TimeTaggerBase*

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

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

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

isConnected()

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

Returns

True/False

Return_type

bool

setDelayClient(channel, delay)

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

Parameters

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

getDelayClient(channel)

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

Parameters

channel (*int*) – Channel number

Returns

input delay in picoseconds

Return_type

int

clearOverflowsClient()

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

getOverflowsClient()

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

Returns

The value of the client-side overflow counter.

Return_type

int

getOverflowsAndClearClient()

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

7.3.5 Additional classes

class SoftwareClockState

The *SoftwareClockState* object contains the current configuration state:

clock_period: int

The rounded clock period matching the input frequency set in `TimeTaggerBase.setSoftwareClock()`.

input_channel: int

The physical input channel of the software clock set in `TimeTaggerBase.setSoftwareClock()`.

ideal_clock_channel: int

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

averaging_periods: float

The averaging periods set in `TimeTaggerBase.setSoftwareClock()`.

enabled: bool

Indicates whether the software clock is active or not.

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

is_locked: bool

Indicates whether the PLL of the software clock was able to lock to the input signal.

error_counter: int

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

last_ideal_clock_event: int

Timestamp of the last ideal clock event in picoseconds.

period_error: float

Current deviation of the measured clock period from the ideal period given by `clock_period`.

phase_error_estimation: float

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

7.4 Virtual Channels

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

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

7.4.1 Available virtual channels

Note: In MATLAB, the Virtual Channel names have common prefix TT*. For example: `Combiner` is named as `TTCombiner`. This prevents possible name collisions with existing MATLAB or user functions.

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.

Combinations

Detects coincidence clicks on all possible combinations of given channels within a given time window, preceded and followed by two guard windows of the same width without any events on these channels.

Combiner

Combines two or more channels into one.

ConstantFractionDiscriminator

Detects rising and falling edges of an input pulse and returns the average time.

DelayedChannel

Clones input channels which can be delayed.

EventGenerator

Generates a signal pattern for every trigger signal.

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.

TriggerOnCountRate

Generates an event when the count rate of a given channel crosses given threshold value.

7.4.2 Common methods

`VirtualChannel.getChannel()`

`VirtualChannel.getChannels()`

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

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

`getConfiguration()`

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

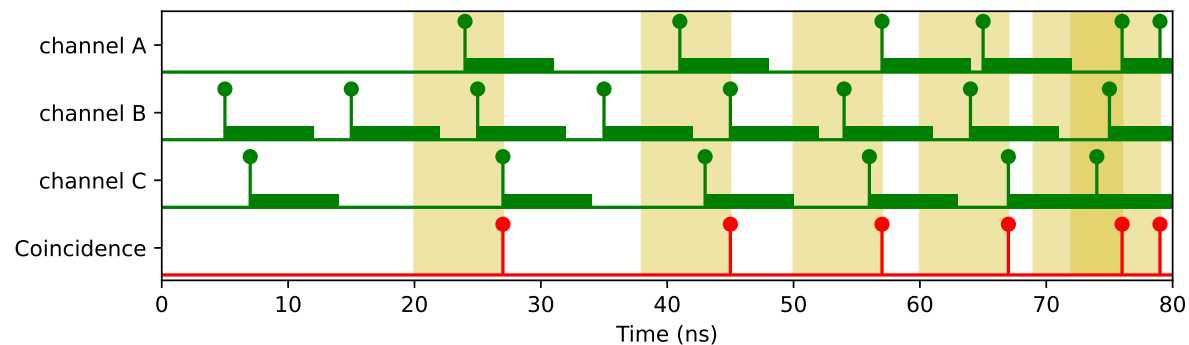
Returns

Configuration data of the virtual channel object.

Return type

dict

7.4.3 Coincidence



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

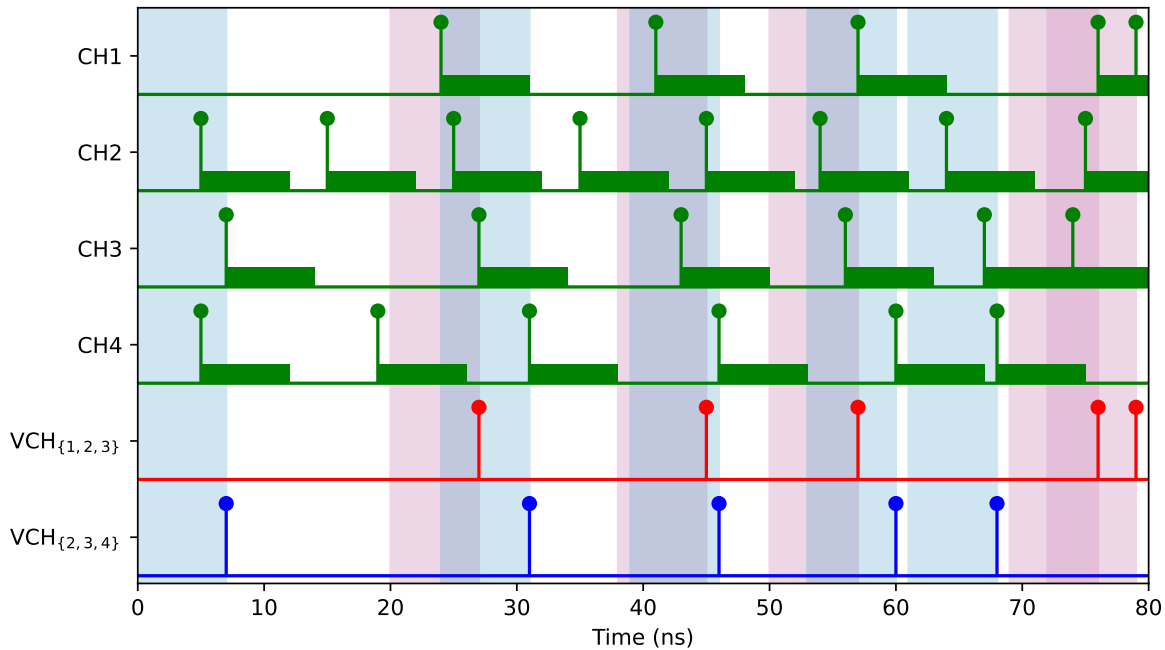
class `Coincidence`(*tagger*, *channels*, *coincidenceWindow*=1000, *timestamp*=`CoincidenceTimestamp.Last`)

Parameters

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

See all common methods

7.4.4 Coincidences



Detects coincidence clicks on multiple channel groups within a given window. If several different coincidences are required with the same window size, *Coincidences* provides better performance in comparison to multiple virtual *Coincidence* channels. One object of the *Coincidence* class is limited to 64 unique channels in the list of channel groups (coincidenceGroup).

Example code:

```
from TimeTagger import Coincidence, Coincidences, CoincidenceTimestamp, createTimeTagger
tagger = createTimeTagger()

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

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

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

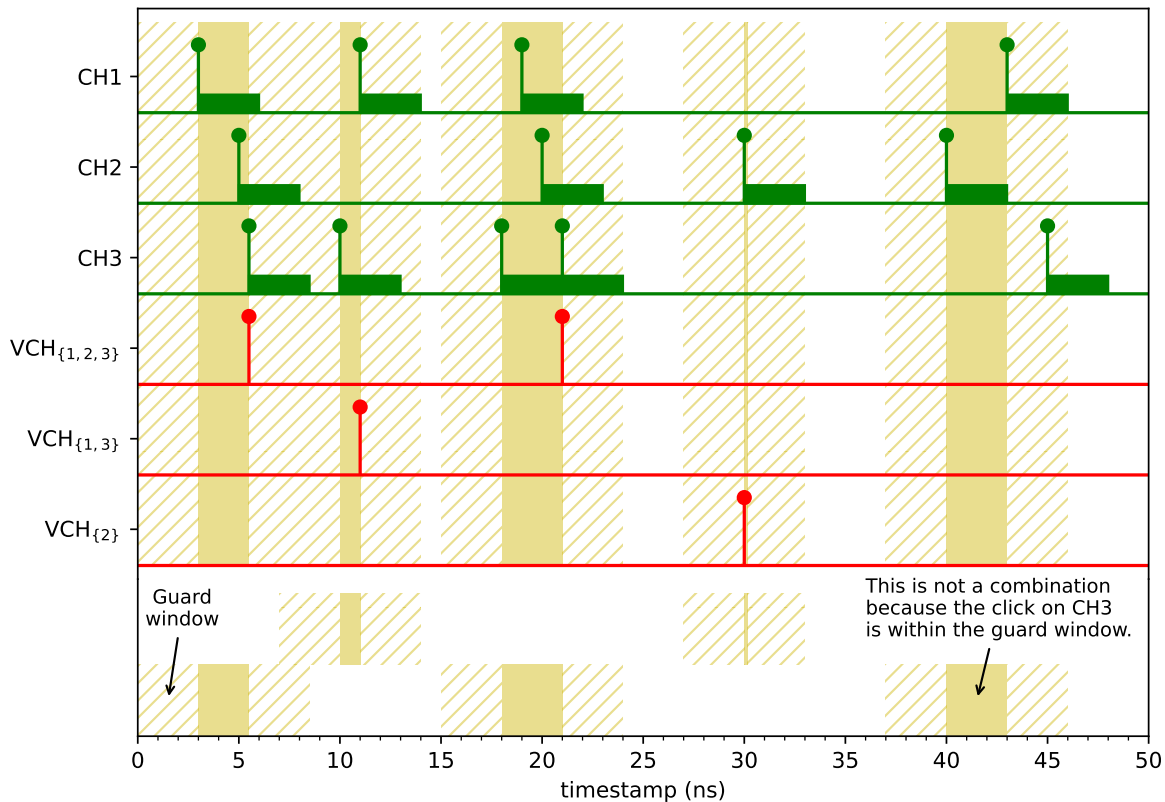
class `Coincidences`(*tagger*, *coincidenceGroups*, *coincidenceWindow*, *timestamp*)

Parameters

- **tagger** (`TimeTaggerBase`) – Time Tagger object instance
- **coincidenceGroups** (`list[list[int]]`) – list of channel groups on which coincidence will be detected in the virtual channel
- **coincidenceWindow** (`int`) – maximum time between all events for a coincidence [ps]
- **timestamp** (`CoincidenceTimestamp`) – type of timestamp for virtual channel (Last, Average, First, ListedFirst)

See all common methods

7.4.5 Combinations



A combination is a group of clicks on a set of channels within a given time window. This time window is surrounded by two guard windows of the same width. These guard windows do not contain any events on the channels being monitored.

The heralding guard window precedes the first click in the combination. The following guard window starts at the time of the last event within the combination window. If there is a click on one of the monitored channels within the guard windows, no combination event is generated. A new combination window then starts with the next click after an empty guard window.

Every time a combination is detected on the monitored channels, Combinations emits a tag on the corresponding virtual channel. The timestamp on this virtual channel is the time of the last event included in the combination. Given N input channels to be monitored, there will be $2^N - 1$ possible combinations, each having a corresponding virtual channel number.

In addition, N extra virtual channels called ‘SumChannels’ are created. This module emits a click on the ‘n-th’ of these channels on each ‘n-fold combination’, regardless of the channels that contributed to the combination. For instance, this is useful for pseudo-photon-number-resolution with detector arrays.

Note: Multiple events on the same channel within one time window are counted as one.

class `Combinations`(*tagger*, *channels*, *window_size*)

Parameters

- **tagger** (`TimeTaggerBase`) – Time Tagger object instance
- **channels** (`list[int]`) – list of channels on which the combinations will be detected
- **window_size** (`int`) – maximum time between all events to make a combination, minimum time without any event detected before and after the combination window [ps]

See all common methods

getChannel(*input_channels*)

Returns the virtual channel number corresponding to the combination formed by the given set of input channels.

Parameters

input_channels (`list[int]`) – list of channels forming the combination monitored by the returned virtual channel

getCombination(*virtual_channel*)

Returns the set of input channels forming a combination event on the given virtual channel *virtual_channel*.

Parameters

virtual_channel (`int`) – virtual channel storing the clicks from the combination formed by the returned channels

Returns

List of channels forming the combination monitored by the input virtual channel

Return type

`list[int]`

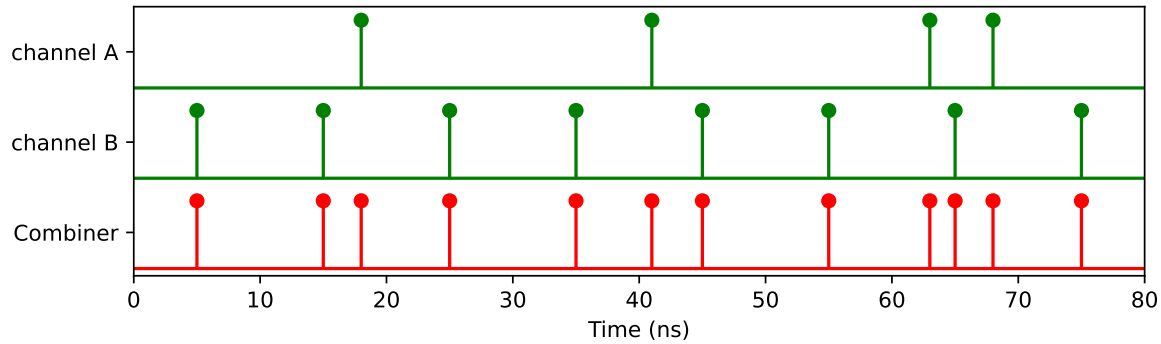
getSumChannel(*n_channels*)

Returns the virtual channel number on which an event is generated when any combination of exactly *n_channels* clicks is detected within the *window_size*.

Parameters

n_channels (`int`) – length of the combinations monitored by the returned virtual channel

7.4.6 Combiner



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

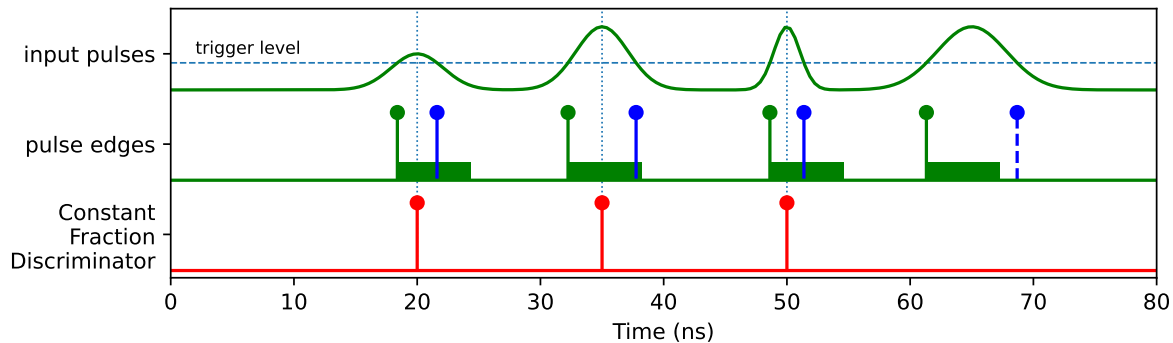
```
class Combiner(tagger, channels=[])
```

Parameters

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

See all common methods

7.4.7 ConstantFractionDiscriminator



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

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

Note: The virtual CFD requires the time tags of the **rising** and **falling** edge. This leads to:

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

In addition, you may encounter data rate limitations due the computational complexity of this virtual channel. Consider using `xtra_setAvgRisingFalling()` for similar functionality when the variation between pulse durations is small. There, the computations are performed on the Time Tagger hardware instead of on your PC, and only half the data rate needs to be transferred for the same result.

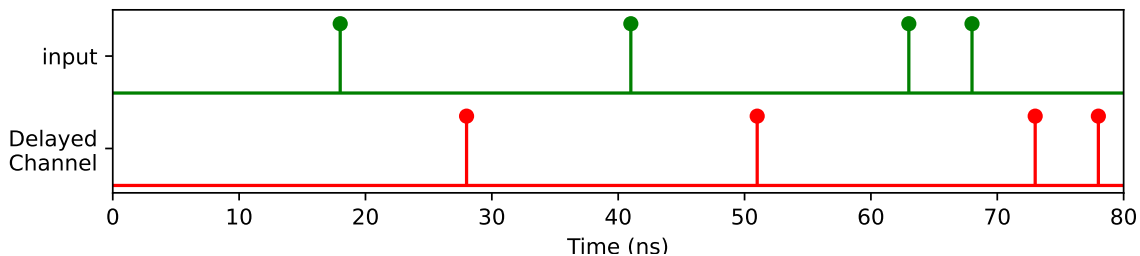
class `ConstantFractionDiscriminator`(*tagger, channels, search_window*)

Parameters

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

See all common methods

7.4.8 DelayedChannel



Clones input channels, which can be delayed by a time specified with the *delay* parameter in the constructor or the `setDelay()` method. A negative delay will delay all other events.

Note: If you want to set a global delay for one or more input channels, `setInputDelay()` is recommended as long as the delays are small, which means that not more than 100 events on all channels should arrive within the maximum delay set.

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

Parameters

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

See all common methods

setDelay(*delay*)

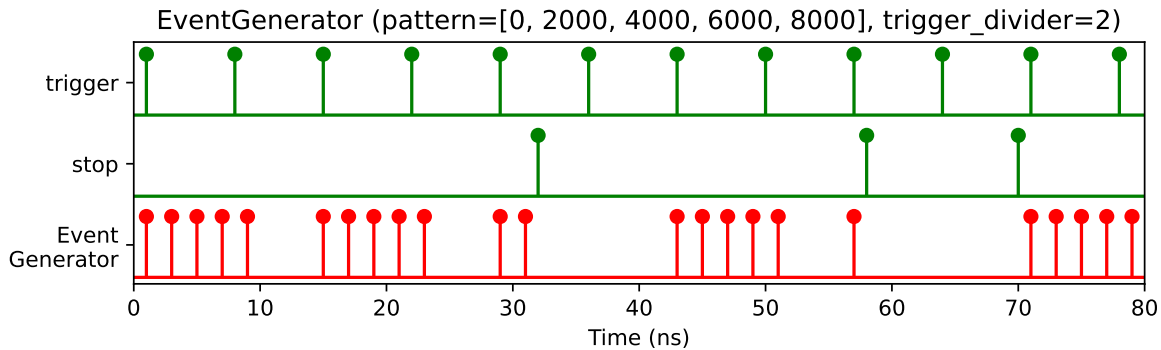
Allows modifying the delay time.

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

Parameters

delay (*int*) – Delay time in picoseconds

7.4.9 EventGenerator



Emits an arbitrary pattern of timestamps for every trigger event. The number of trigger events can be reduced by *trigger_divider*. The start of a new pattern does not abort the execution of unfinished patterns, so patterns may overlap. The execution of all running patterns can be aborted by a click of the *stop_channel*, i.e. overlapping patterns can be avoided by setting the *stop_channel* to the *trigger_channel*.

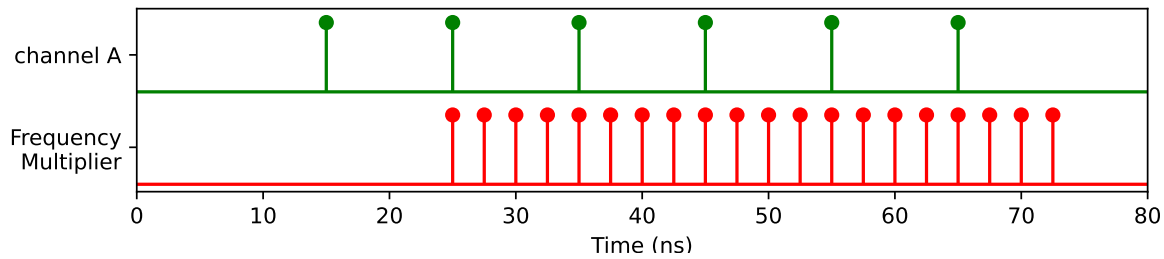
class EventGenerator(*tagger*, *trigger_channel*, *pattern*, *trigger_divider*, *stop_channel*)

Parameters

- **tagger** (*TimeTaggerBase*) – Time Tagger object instance.
- **trigger_channel** (*int*) – Channel number of the trigger signal.
- **pattern** (*list[int]*) – List of relative timestamps defining the pattern executed upon a trigger event.
- **trigger_divider** (*int*) – Factor by which the number of trigger events is reduced. (default: 1)
- **divider_offset** (*int*) – If *trigger_divider* > 1, the *divider_offset* the number of trigger clicks to be ignored before emitting the first pattern. (default: 0)
- **stop_channel** (*int*) – Channel number of the stop channel. (optional)

See all common methods

7.4.10 FrequencyMultiplier



Frequency Multiplier for a channel with a periodic signal.

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

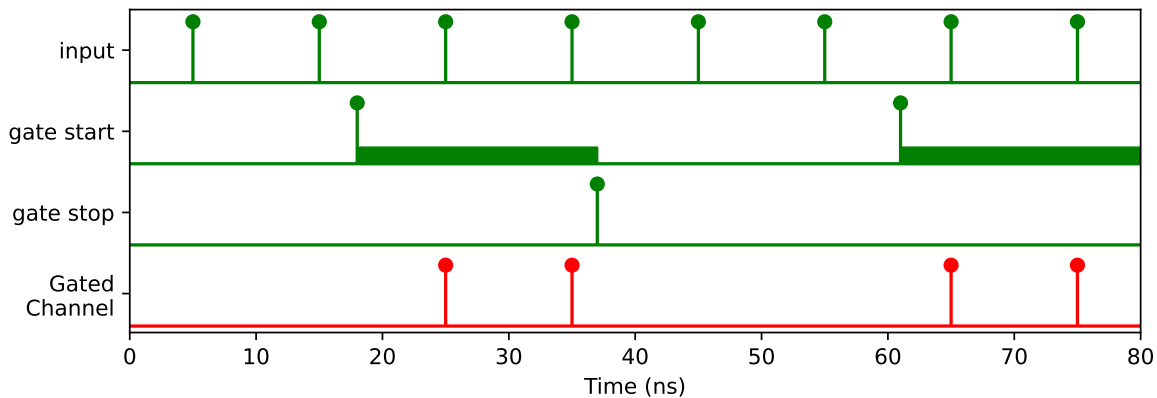
```
class FrequencyMultiplier(tagger, input_channel, multiplier)
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – Time Tagger object instance
- **input_channel** ([int](#)) – channel on which the upscaling of the frequency is based on
- **multiplier** ([int](#)) – frequency upscaling factor

See all common methods

7.4.11 GatedChannel



Transmits the signal from an `input_channel` to a new virtual channel between an edge detected at the `gate_start_channel` and the `gate_stop_channel`.

```
class GatedChannel(tagger, input_channel, gate_start_channel, gate_stop_channel,
                  initial=GatedChannelInitial.Closed)
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – Time Tagger object
- **input_channel** ([int](#)) – Channel which is gated

- **gate_start_channel** (*int*) – Channel on which a signal detected will start the transmission of the input_channel through the gate
- **gate_stop_channel** (*int*) – Channel on which a signal detected will stop the transmission of the input_channel through the gate. Note that `gate_stop_channel == gate_start_channel` will result in an exception.
- **initial** (*GatedChannelInitial*) – The initial state of the gate. If overflows occur, the gate will be reset to this state as well. By default, the state is *Closed*.

See all common methods

Note: If you assign the same channel to `input_channel` and to `gate_start_channel` or `gate_stop_channel`, respectively, the internal execution order of the transmission decision and the gate operation (opening or closing) becomes important: For each tag on the `input_channel`, the decision is made based on the previous state. After this decision is made for itself, the tag might toggle the gate state.

- **`input_channel == gate_stop_channel:`**

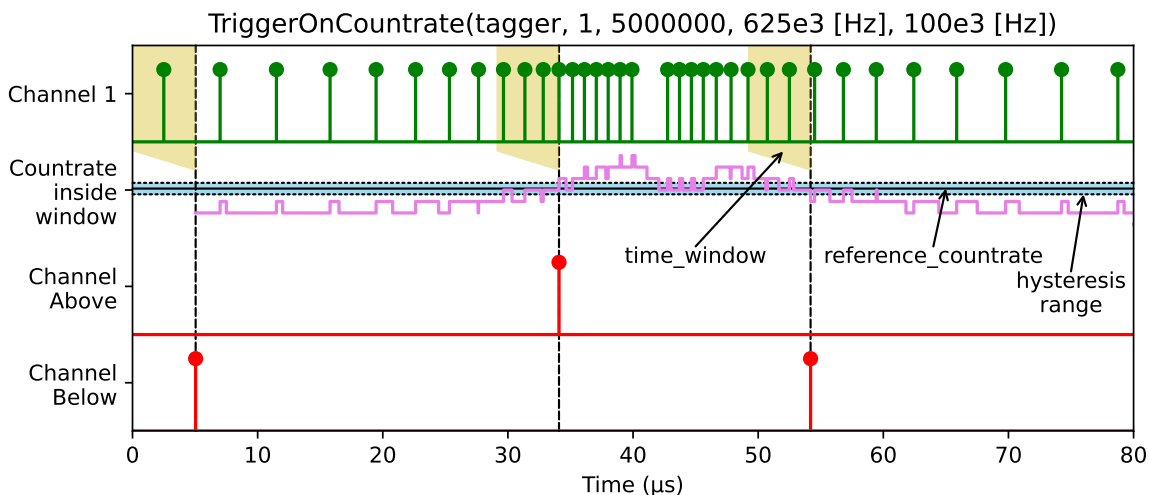
If the gate is open prior to the arrival of the tag, the tag will pass the gate and close it afterward. All subsequent tags will be eliminated until an event on `gate_start_channel` opens the gate again. This means that after the gate has been opened, only a single tag will pass the gate, which is exactly the behavior of the Conditional Filter with `gate_start_channel` acting as the trigger and `input_channel` acting as the filtered channel.

- **`input_channel == gate_start_channel:`**

If the gate is open prior to the arrival of the tag, the tag itself will be blocked but opens the gate afterward. All subsequent tags will pass the gate until an event on `gate_stop_channel` closes the gate again. This means that every event on `gate_stop_channel` will eliminate exactly the next event on the `input_channel`.

This behavior has been changed in software version 2.10.8.

7.4.12 TriggerOnCountrate



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

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

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

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

class TriggerOnCountrate(tagger, input_channel, reference_countrate, hysteresis, time_window)

Parameters

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

See all common methods

getChannelAbove()

Get the channel number of the *above* channel.

getChannelBelow()

Get the channel number of the *below* channel.

getChannels()

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

getCurrentCountrate()

Get the current count rate averaged within the *time_window*.

injectCurrentState()

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

isAbove()

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

isBelow()

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

7.5 Measurement Classes

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

All measurements provide a set of methods to start and stop the execution and to access the accumulated data. In a typical application, the following steps are performed (see *example*):

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

7.5.1 Available measurement classes

Note: In MATLAB, the Measurement names have common prefix `TT*`. For example: `Correlation` is named as `TTCorrelation`. This prevents possible name collisions with existing MATLAB or user functions.

Correlation

Auto- and Cross-correlation measurement.

CountBetweenMarkers

Counts tags on one channel within bins which are determined by triggers on one or two other channels. Uses a static buffer output. Use this to implement a gated counter, a counter synchronized to external signals, etc.

Counter

Counts the clicks on one or more channels with a fixed bin width and a circular buffer output.

Countrate

Average tag rate on one or more channels.

Dump

Deprecated - please use *FileWriter* instead. Dump measurement writes all time-tags into a file.

FileReader

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

FileWriter

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

Flim

Fluorescence lifetime imaging.

FrequencyCounter

Analyze the phase evolution of multiple channels at periodic sampling points.

FrequencyStability

Analyzes the frequency stability of period signals.

Histogram

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

Histogram2D

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

HistogramLogBins

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

HistogramND

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

IteratorBase

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

PulsePerSecondMonitor

Monitors the synchronicity of 1 pulse per second (PPS) signals.

Sampler

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

Scope

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

StartStop

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

SynchronizedMeasurements

Helper class that allows synchronization of the measurement classes.

TimeDifferences

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

TimeDifferencesND

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

TimeTagStream

This class provides you with access to the time-tag stream and allows you to implement your own on-the-fly processing. See [Raw Time-Tag-Stream access](#) to get an overview about the possibilities for the raw time-tag-stream access.

7.5.2 Common methods

class *IteratorBase*

clear()

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

start()

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

startFor(*duration*[, *clear=True*])

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

stop()

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

abort()

Immediately aborts the measurement, discards accumulated measurement data, and resets the state to the initial state.

Warning: After calling `abort()`, the last block of data might become irreversibly corrupted. Please always use `stop()` to end a measurement.

isRunning()

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

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

Returns

True/False

Return type

bool

waitUntilFinished(timeout=-1)

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

This is roughly equivalent to a polling loop with `sleep()`.

```
measurement.waitUntilFinished(timeout=-1)
# is roughly equivalent to
while measurement.isRunning():
    sleep(0.01)
```

Parameters

timeout (*int*) – timeout in milliseconds. Negative value means no timeout, zero returns immediately.

Returns

True if the measurement has finished, False on timeout

Return type

bool

getCaptureDuration()

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

Returns

Capture duration in ps

Return type

int

getConfiguration()

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

Returns

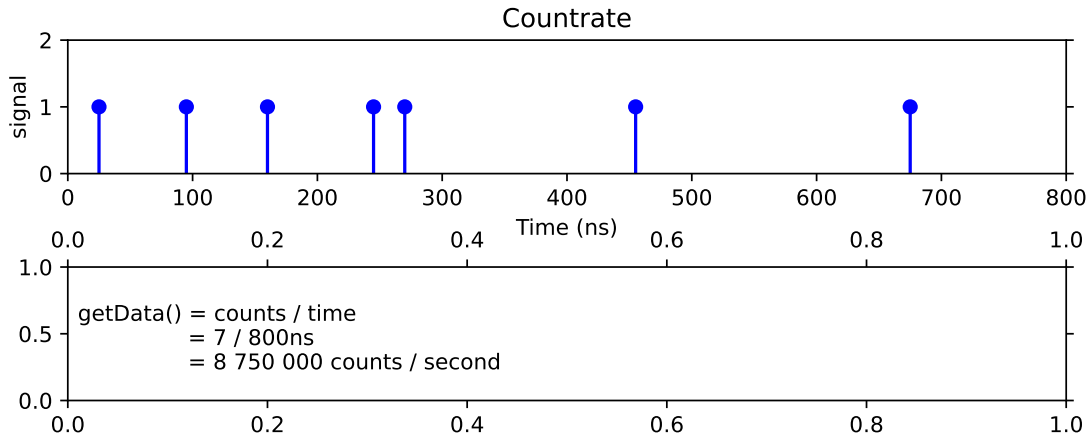
Configuration data of the measurement object.

Return type

`dict`

7.5.3 Event counting

Countrate



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

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

Parameters

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

See all common methods

getData()

Returns

Average count rate in counts per second.

Return type

`1D_array[float]`

getCountsTotal()

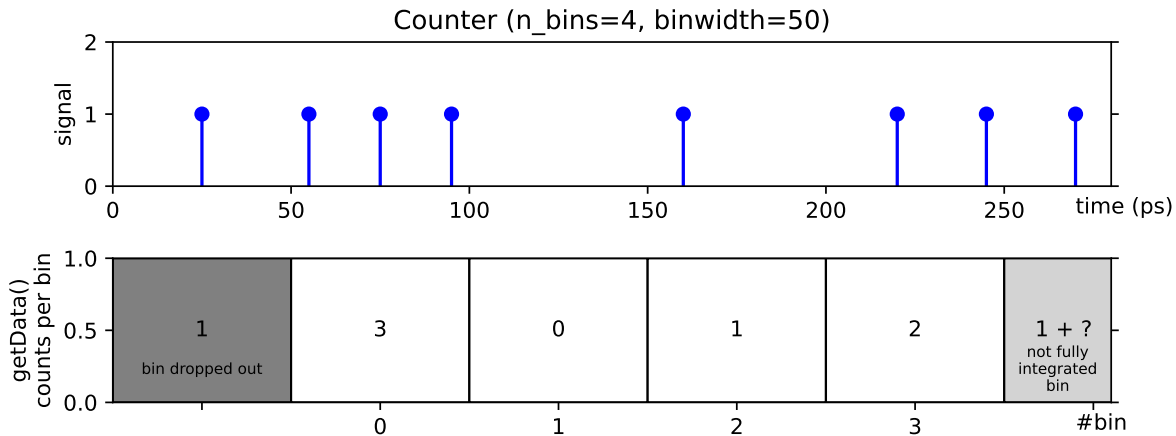
Returns

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

Return type

`1D_array[int]`

Counter



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

class Counter(*tagger*, *channels*[, *binwidth*=1e9, *n_values*=1])

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object
- **channels** ([list](#)[[int](#)]) – channels used for counting tags
- **binwidth** ([int](#)) – bin width in ps (default: 1e9)
- **n_values** ([int](#)) – number of bins (default: 1)

See all common methods

getData([*rolling*=True])

Returns an array of accumulated counter bins for each channel. The optional parameter *rolling*, controls if the not integrated bins are padded before or after the integrated bins.

When *rolling*=True, the most recent data is stored in the last bin of the array and every new completed bin shifts all other bins right-to-left. When continuously plotted, this creates an effect of rolling trace plot. For instance, it is useful for continuous monitoring of countrate changes over time.

When *rolling*=False, the most recent data is stored in the next bin after previous such that the array is filled up left-to-right. When array becomes full and the Counter is still running, the array index will be reset to zero and the array will be filled again overwriting previous values. This operation is sometimes called “sweep plotting”.

Parameters

rolling ([bool](#)) – Controls how the counter array is filled.

Returns

An array of size ‘number of channels’ by *n_values* containing the counts in each fully integrated bin.

Return type

2D_array[int]

getIndex()

Returns the relative time of the bins in ps. The first entry of the returned vector is always 0.

Returns

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

Return type

1D_array[int]

getDataNormalized([rolling=True])

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

Return type

2D_array[float]

getDataTotalCounts()

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

Returns

Number of events per channel.

Return type

1D_array[int]

getDataObject(remove=False)

Returns [CounterData](#) object containing a snapshot of the data accumulated in the [Counter](#) at the time this method is called.

Parameters

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

Returns

An object providing access to a snapshot data.

Return type[CounterData](#)**class CounterData**

Objects of this class are created and returned by [Counter.getDataObject\(\)](#), and contain a snapshot of the data accumulated by the [Counter](#) measurement.

size: *int*

Number of returned bins.

dropped_bins: *int*

Number of bins which have been dropped because *n_values* of the [Counter](#) has been exceeded.

overflow: *bool*

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

getIndex()

Returns the relative time of the bins in ps. The first entry of the returned vector is always 0 for *size* > 0.

Returns

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

Return type

1D_array[int]

getData()**Returns**An array of size 'number of channels' by *size* containing only fully integrated bins.**Return type**

2D_array[int]

getDataNormalized()

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

Return type

2D_array[float]

getFrequency(time_scale: int = 1_000_000_000_000)

Returns the counts normalized to the specified time scale. Bins in overflow mode are marked as *NaN*.

Parameters

time_scale – Scales the return value to this time interval. Default is 1 s, so the return value is in Hz. For negative values, the time scale is set to *binwidth*.

Return type

2D_array[float]

getDataTotalCounts()

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

Returns

Number of events per channel.

Return type

1D_array[int]

getTime()

This is similar to *getIndex()* but it returns the absolute timestamps of the bins. For subsequent calls to *getDataObject()*, these arrays can be concatenated to obtain a full index array.

Returns

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

Return type

1D_array[int]

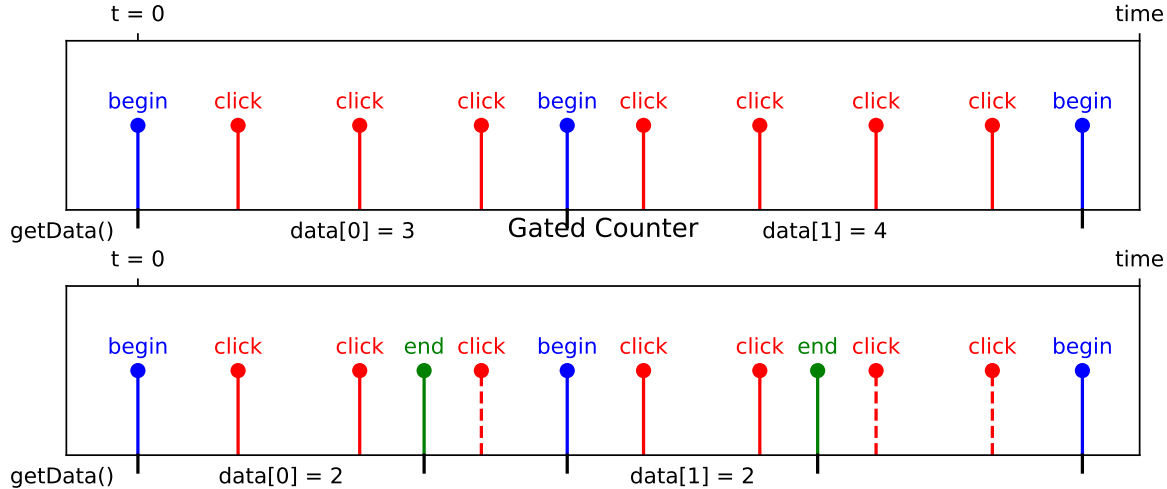
getOverflowMask()

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

ReturnsAn array of size *size* containing overflow mask.**Return type**

1D_array[int]

CountBetweenMarkers



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

```
class CountBetweenMarkers(tagger, click_channel, begin_channel[, end_channel=CHANNEL_UNUSED,
n_values=1000 ])
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object
- **click_channel** (*int*) – channel on which clicks are received, gated by *begin_channel* and *end_channel*
- **begin_channel** (*int*) – channel that triggers the beginning of counting and stepping to the next value
- **end_channel** (*int*) – channel that triggers the end of counting (optional, default: [CHANNEL_UNUSED](#))
- **n_values** (*int*) – number of values stored (data buffer size) (default: 1000)

See all common methods

getData()

Returns

Array of size n_values containing the acquired counter values.

Return type

1D_array[*int*]

getIndex()**Returns**

Vector of size n_values containing the time in ps of each start click in respect to the very first start click.

Return type

1D_array[int]

getBinWidths()**Returns**

Vector of size n_values containing the time differences of 'start -> (next start or stop)' for the acquired counter values.

Return type

1D_array[int]

ready()**Returns**

True when the entire array is filled.

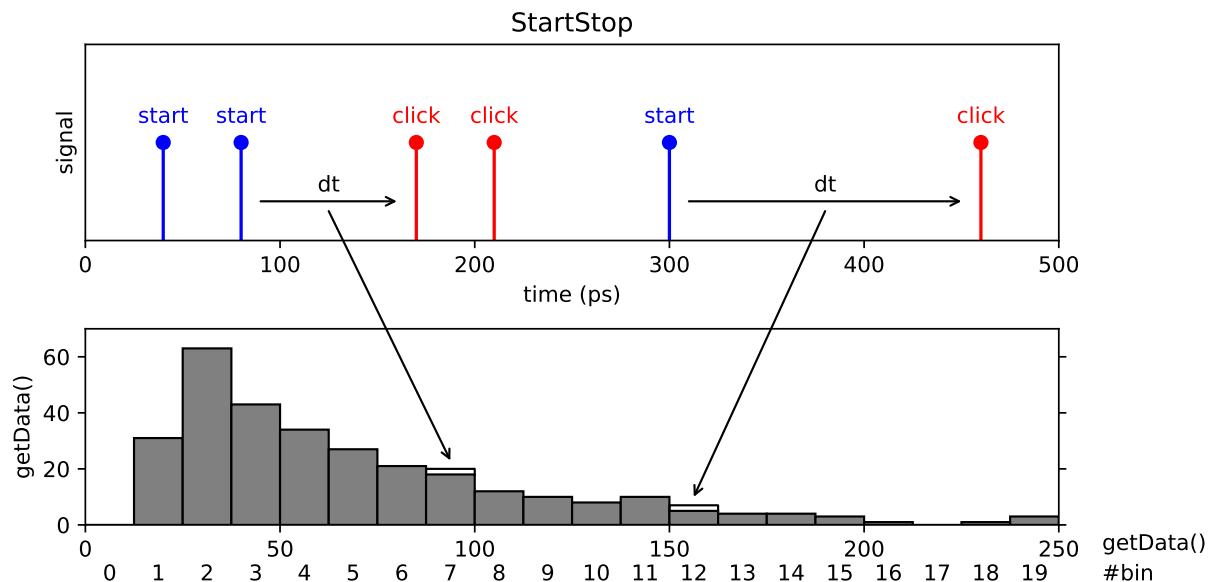
Return type

bool

7.5.4 Time histograms

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

StartStop



A simple start-stop measurement. This class performs a start-stop measurement between two channels and stores the time differences in a histogram. The histogram resolution is specified beforehand (*binwidth*) but the histogram range

(number of bins) is unlimited. It is adapted to the largest time difference that was detected. Thus, all pairs of subsequent clicks are registered. Only non-empty bins are recorded.

```
class StartStop(tagger, click_channel[, start_channel=CHANNEL_UNUSED, binwidth=1000 ])
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object instance
- **click_channel** (*int*) – channel on which stop clicks are received
- **start_channel** (*int*) – channel on which start clicks are received (default: [CHANNEL_UNUSED](#))
- **binwidth** (*int*) – bin width in ps (default: 1000)

See all common methods

getData()

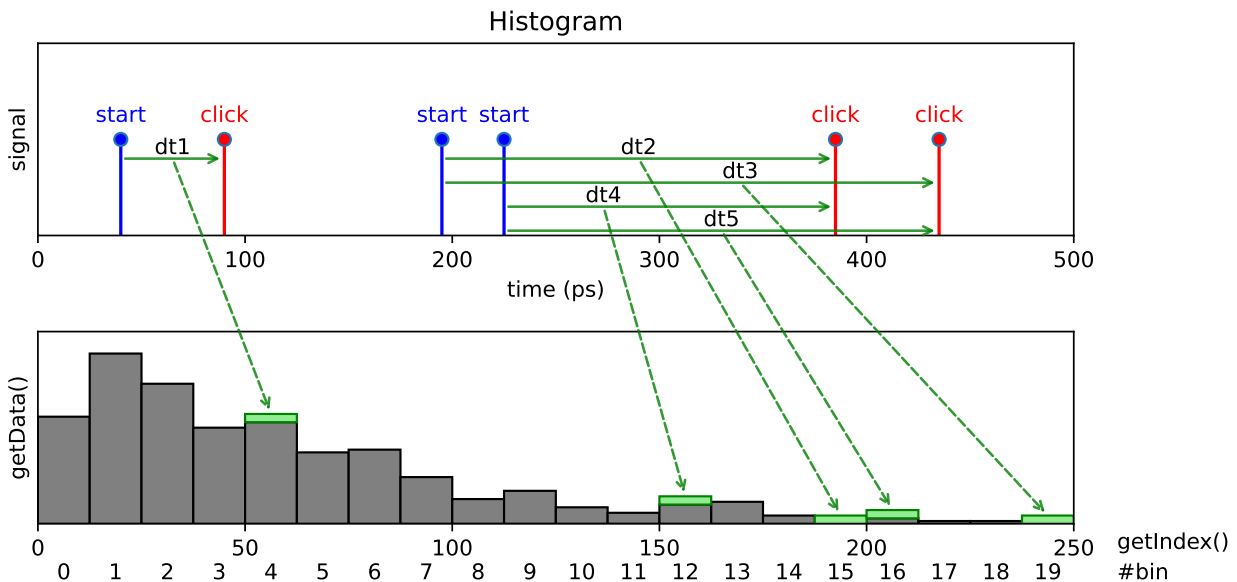
Returns

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

Return type

2D_array[*int*]

Histogram



Accumulate time differences into a histogram. This is a simple multiple start, multiple stop measurement. This is a special case of the more general [TimeDifferences](#) measurement. Specifically, the measurement waits for clicks on the *start_channel*, and for each start click, it measures the time difference between the start clicks and all subsequent clicks on the *click_channel* and stores them in a histogram. The histogram range and resolution are specified by the number of bins and the bin width specified in ps. Clicks that fall outside the histogram range are ignored. Data accumulation is performed independently for all start clicks. This type of measurement is frequently referred to as a 'multiple start, multiple stop' measurement and corresponds to a full auto- or cross-correlation measurement.

```
class Histogram(tagger, click_channel[, start_channel=CHANNEL_UNUSED, binwidth=1000, n_bins=1000 ])
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object instance
- **click_channel** ([int](#)) – channel on which clicks are received
- **start_channel** ([int](#)) – channel on which start clicks are received (default: [CHANNEL_UNUSED](#))
- **binwidth** ([int](#)) – bin width in ps (default: 1000)
- **n_bins** ([int](#)) – the number of bins in the histogram (default: 1000)

See all common methods

getData()**Returns**

A one-dimensional array of size *n_bins* containing the histogram.

Return type

1D_array[[int](#)]

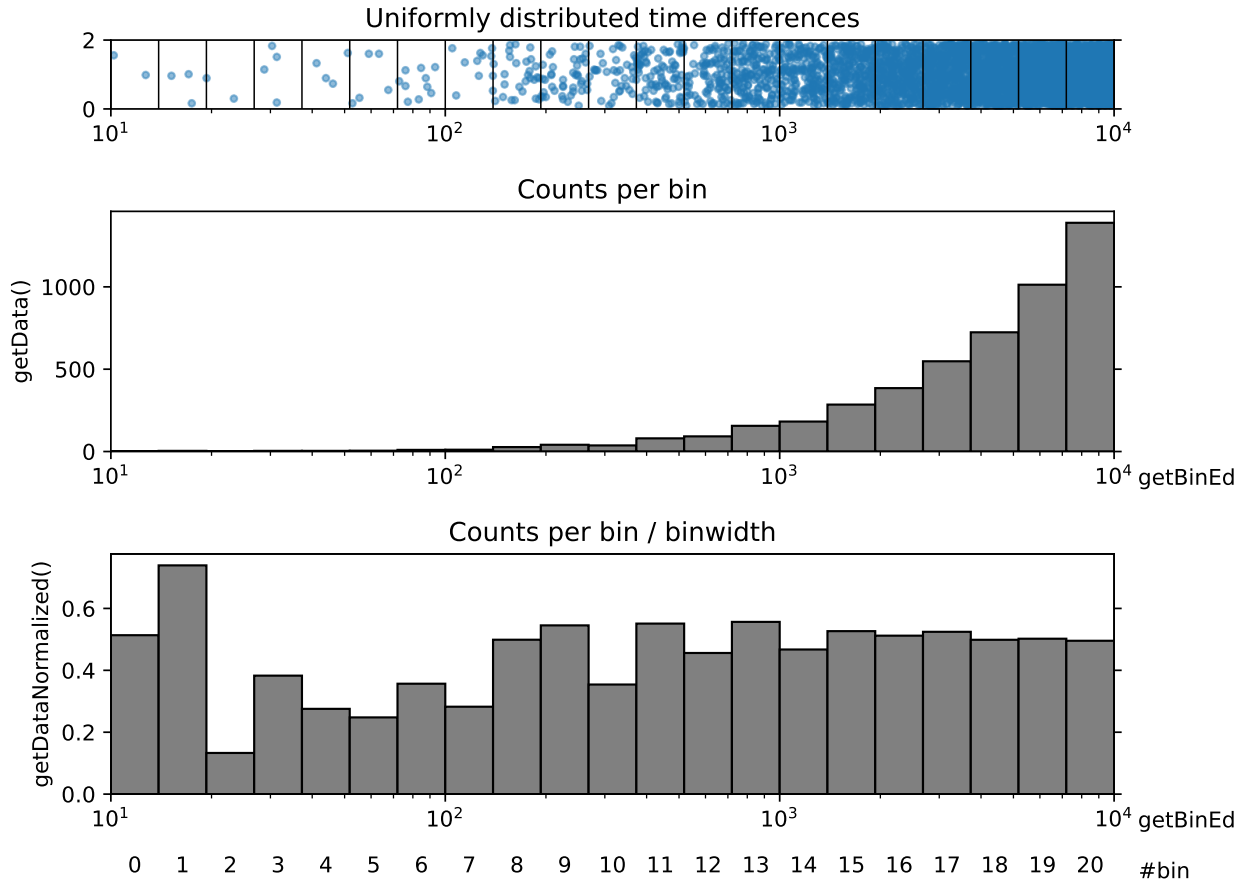
getIndex()**Returns**

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

Return type

1D_array[[int](#)]

HistogramLogBins



The HistogramLogBins measurement is similar to [Histogram](#) but the bin edges are spaced logarithmically. As the bins do not have a homogeneous binwidth, a proper normalization is required to interpret the raw data.

For excluding time ranges from the histogram evaluation while maintaining a proper normalization, [HistogramLogBins](#) optionally takes two gating arguments of type [ChannelGate](#). This can, e.g., be used to pause the acquisition during erroneous ranges that have to be identified by virtual channels. The same mechanism automatically applies to overflow ranges.

The acquired histogram $H(t)$ is normalized by

$$\tilde{H}(\tau) = I(\tau) \cdot C_{\text{click}} \cdot C_{\text{start}}$$

with an estimation of counts $I(\tau)$ and the click and start channel count rates, $C_{\text{click}} = N_{\text{click}}/t_{\text{click}}$ and $C_{\text{start}} = N_{\text{start}}/t_{\text{start}}$, respectively. Typically, this will be used to calculate

$$g^{(2)}(\tau) = \frac{H(\tau)}{\tilde{H}(\tau)}$$

For $t \gg 10^{\text{exp_stop}}$ s and without interruptions, $I(\tau)/t$ will approach the binwidth of the respective bin. During the early acquisition and in case of interruptions, $I(\tau)$ can be significantly smaller, which compensates for counts that are excluded from $H(\tau)$.

```
class HistogramLogBins(tagger, click_channel, start_channel, exp_start, ex_stop, n_bins, click_gate=None,
                        start_gate=None)
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object instance
- **click_channel** ([int](#)) – channel on which clicks are received
- **start_channel** ([int](#)) – channel on which start clicks are received
- **exp_start** ([float](#)) – exponent $10^{\text{exp_start}}$ in seconds where the very first bin begins
- **exp_stop** ([float](#)) – exponent $10^{\text{exp_stop}}$ in seconds where the very last bin ends
- **n_bins** ([int](#)) – the number of bins in the histogram
- **click_gate** ([ChannelGate](#)) – optional evaluation gate for the *click_channel*
- **start_gate** ([ChannelGate](#)) – optional evaluation gate for the *start_channel*

See all common methods

getDataObject()**Returns**

A data object containing raw and normalization data.

Return type

[HistogramLogBinsData](#)

getBinEdges()**Returns**

A vector of size n_bins+1 containing the bin edges in picoseconds.

Return type

`1D_array[int]`

getData()

Deprecated since version 2.17.0: please use [HistogramLogBins.getDataObject\(\)](#) and [HistogramLogBinsData.getCounts\(\)](#) instead.

Returns

A one-dimensional array of size n_bins containing the histogram.

Return type

`1D_array[int]`

getDataNormalizedCountsPerPs()

Deprecated since version 2.17.0.

Returns

The counts normalized by the binwidth of each bin.

Return type

`1D_array[float]`

getDataNormalizedG2()

Deprecated since version 2.17.0: please use [getDataObject\(\)](#) and [HistogramLogBinsData.getG2\(\)](#) instead.

Returns

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

Return type

`1D_array[float]`

class HistogramLogBinsData

Contains the histogram counts $H(\tau)$ and the corresponding normalization function $\tilde{H}(\tau)$.

getG2()**Returns**

A one-dimensional array of size n_bins containing the normalized histogram $H(\tau)/\tilde{H}(\tau)$.

Return type

1D_array[float]

getCounts()**Returns**

A one-dimensional array of size n_bins containing the raw histogram $H(\tau)$.

Return type

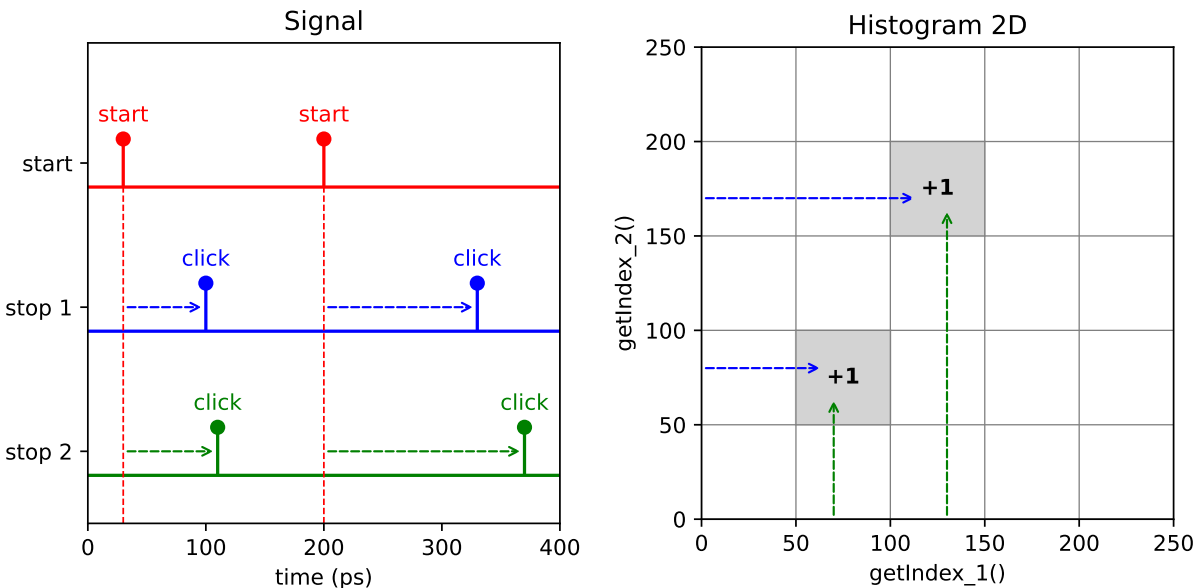
1D_array[int]

getG2Normalization()**Returns**

A one-dimensional array of size n_bins containing the normalization $\tilde{H}(\tau)$.

Return type

1D_array[float]

Histogram2D

This measurement is a 2-dimensional version of the *Histogram* measurement. The measurement accumulates a two-dimensional histogram where stop signals from two separate channels define the bin coordinate. For instance, this kind of measurement is similar to that of typical 2D NMR spectroscopy. The data within the histogram is acquired via a single-start, single-stop analysis for each axis. The first stop click of each axis is taken after the start click to evaluate the histogram counts.


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

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object
- **start_channel** ([int](#)) – channel on which start clicks are received
- **stop_channel_1** ([int](#)) – channel on which stop clicks for the time axis 1 are received
- **stop_channel_2** ([int](#)) – channel on which stop clicks for the time axis 2 are received
- **binwidth_1** ([int](#)) – bin width in ps for the time axis 1
- **binwidth_2** ([int](#)) – bin width in ps for the time axis 2
- **n_bins_1** ([int](#)) – the number of bins along the time axis 1
- **n_bins_2** ([int](#)) – the number of bins along the time axis 2

See all common methods

getData()

Returns

A two-dimensional array of size *n_bins_1* by *n_bins_2* containing the 2D histogram.

Return type

2D_array[[int](#)]

getIndex()

Returns a 3D array containing two coordinate matrices (*meshgrid*) for time bins in ps for the time axes 1 and 2. For details on *meshgrid* please take a look at the respective documentation either for [Matlab](#) or [Python NumPy](#).

Returns

A three-dimensional array of size *n_bins_1* x *n_bins_2* x 2

Return type

3D_array[[int](#)]

getIndex_1()

Returns

A vector of size *n_bins_1* containing the bin locations in ps for the time axis 1.

Return type

1D_array[[int](#)]

getIndex_2()

Returns

A vector of size *n_bins_2* containing the bin locations in ps for the time axis 2.

Return type

1D_array[[int](#)]

HistogramND

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

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

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

Parameters

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

See all common methods

getData()

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

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

Returns

Flattened array of histogram bins.

Return type

1D_array[int]

getIndex(dim)

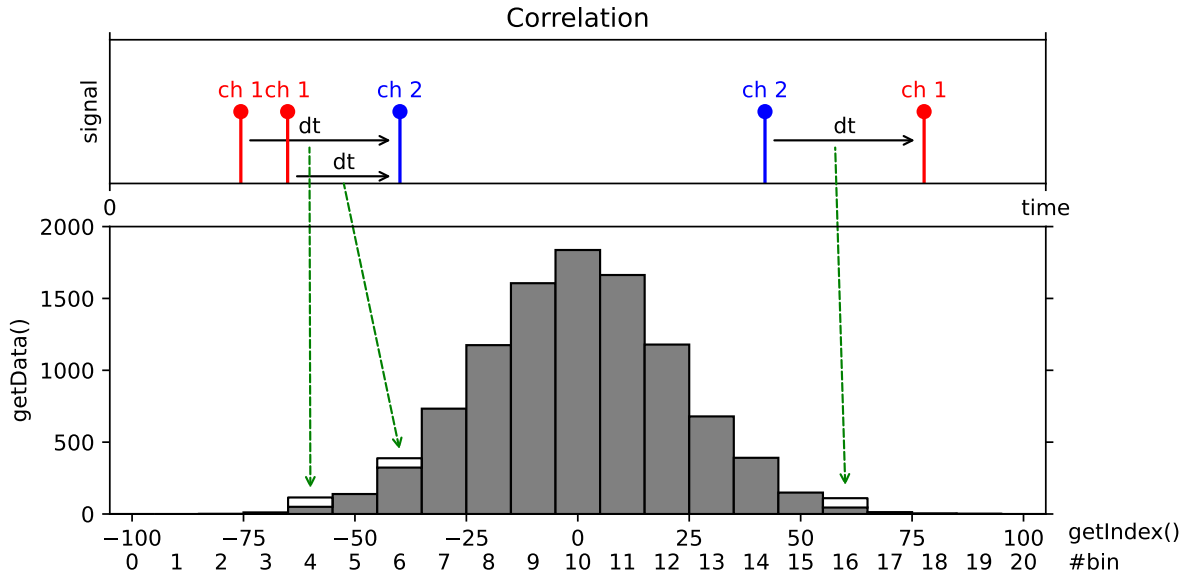
Returns

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

Return type

1D_array[int]

Correlation



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

class Correlation(tagger, channel_1[, channel_2=CHANNEL_UNUSED, binwidth=1000, n_bins=1000])

Parameters

- **tagger** ([TimeTaggerBase](#)) – time tagger object
- **channel_1** (*int*) – channel on which (stop) clicks are received
- **channel_2** (*int*) – channel on which reference clicks (start) are received (when left empty or set to [CHANNEL_UNUSED](#) -> an auto-correlation measurement is performed, which is the same as setting `channel_1 = channel_2`) (default: [CHANNEL_UNUSED](#))
- **binwidth** (*int*) – bin width in ps (default: 1000)
- **n_bins** (*int*) – the number of bins in the resulting histogram (default: 1000)

See all common methods

getData()

Returns

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

Return type

1D_array[[int](#)]

getDataNormalized()

Return the data normalized as:

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

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

Returns

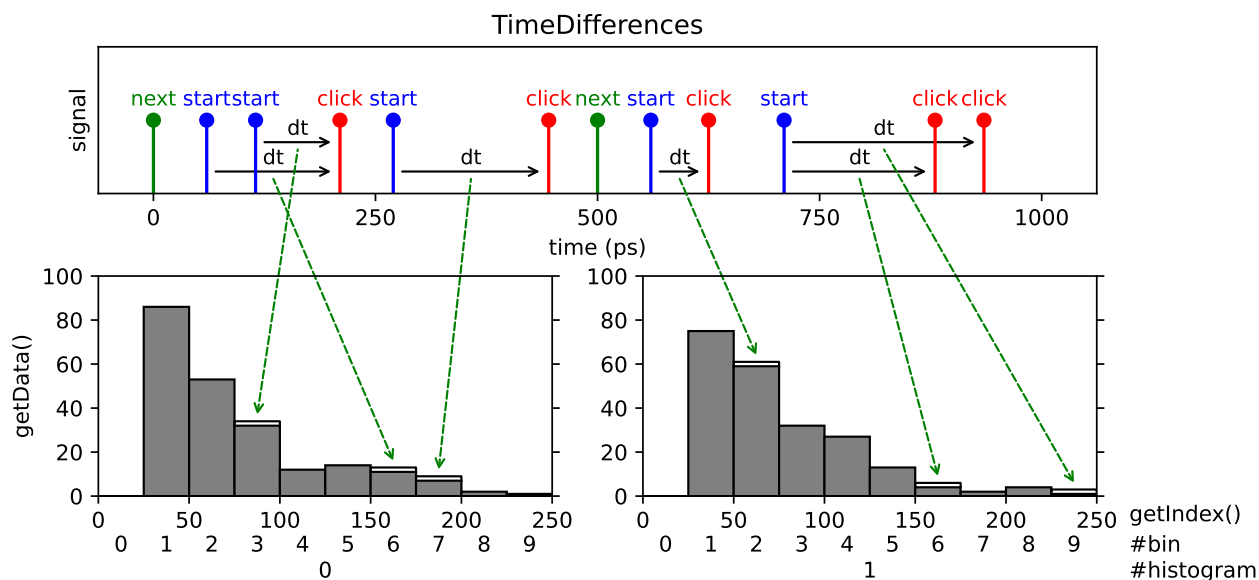
Data normalized by the binwidth and the average count rate.

Return type

1D_array[float]

getIndex()**Returns**A vector of size n_bins containing the time bins in ps.**Return type**

1D_array[int]

TimeDifferences

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

The time-difference data can be accumulated into a single histogram or into multiple subsequent histograms. In this way, you can record a sequence of time-difference histograms. To switch from one histogram to the next one you have to specify a channel that provide switch markers (*next_channel* parameter). Also you need to specify the number of histograms with the parameter *n_histograms*. After each tag on the *next_channel*, the histogram index is incremented by one and reset to zero after reaching the last valid index. The measurement starts with the first tag on the *next_channel*.

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

Typically, you will run the measurement indefinitely until stopped by the user. However, it is also possible to specify the maximum number of rollovers of the histogram index. In this case, the measurement stops when the number of rollovers has reached the specified value.

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

Parameters

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

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

See all common methods

getData()

Returns

A two-dimensional array of size *n_bins* by *n_histograms* containing the histograms.

Return type

2D_array[int]

getIndex()

Returns

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

Return type

1D_array[int]

setMaxCounts()

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

getHistogramIndex()

Returns

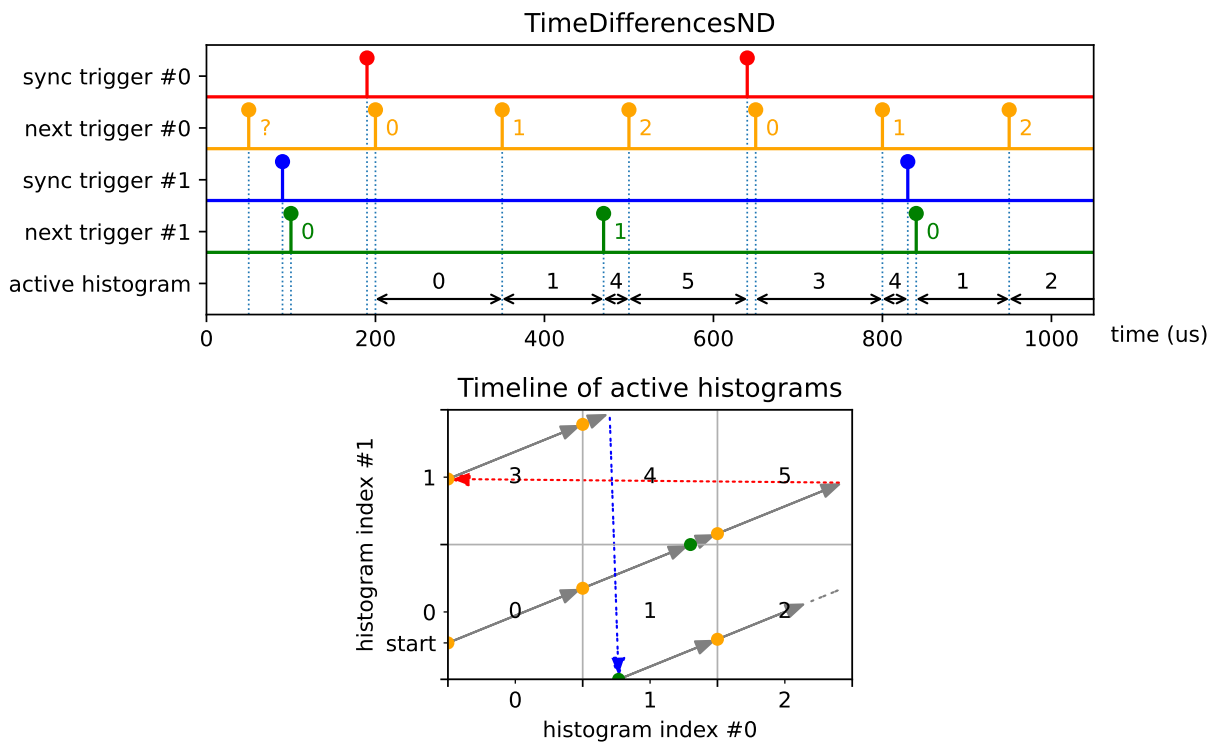
The index of the currently processed histogram or the waiting state. Possible return values are:

- -2: Waiting for an event on *sync_channel* (only if *sync_channel* is defined)

- `-1`: Waiting for an event on `next_channel` (only if `next_channel` is defined)
- `0 ... (n_histograms - 1)`: Index of the currently processed histogram

Return type`int`**getCounts()****Returns**

The number of rollovers (histogram index resets).

Return type`int`**ready()****Returns**True when the required number of rollovers set by `setMaxCounts()` has been reached.**Return type**`bool`**TimeDifferencesND**

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

Please read the documentation of *TimeDifferences* first. It captures many multiple start - multiple stop histograms, but with many asynchronous *next_channel* triggers. After each tag on each *next_channel*, the histogram index of the associated dimension is incremented by one and reset to zero after reaching the last valid index. The elements of the

parameter *n_histograms* specify the number of histograms per dimension. The accumulation starts when *next_channel* has been triggered on all dimensions.

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

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

old start trigger → all sync triggers → all next triggers → new start trigger

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

Parameters

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

See all common methods

See methods of `TimeDifferences` class.

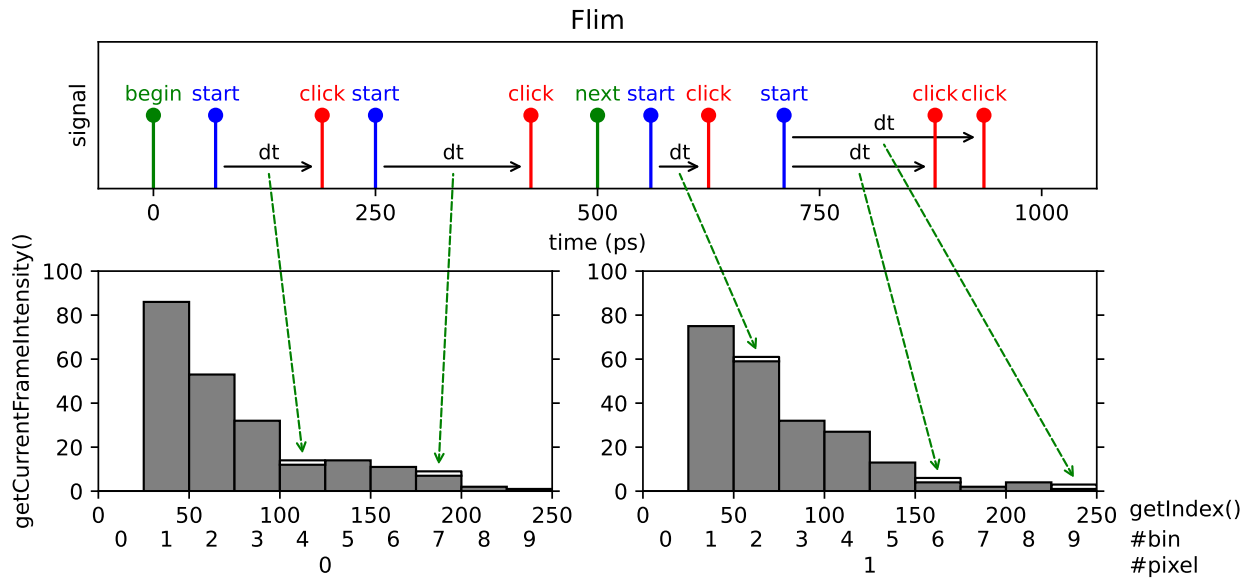
7.5.5 Fluorescence-lifetime imaging (FLIM)

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

Flim

Changed in version 2.7.2.

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



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

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

This measurement implements a line scan in a FLIM 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.

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

Please visit the Python example folder for a reference implementation.

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

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

# FLIM 2.7.0 replacement using TimeDifferences
TimeDifferences(tagger, click_channel=1, start_channel=2,
               next_channel=3, sync_channel=CHANNEL_UNUSED,
               binwidth=100, n_bins=1000, n_histograms=320*240)
```

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

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

This class supports expansion of functionality with custom FLIM frame processing by overriding virtual/abstract *frameReady()* callback. If you need custom implementation with minimal overhead and highest performance, consider overriding *FlimBase* class instead.

Warning: When overriding this class, you must set *pre_initialize=False* and then call *initialize()* at the end of your custom constructor code. Otherwise, you may experience unstable or erratic behavior of your program, as the callback *frameReady()* may be called before construction of the subclass completed.

The data query methods are organized into a few groups.

The methods *getCurrentFrame...()* relate to the active frame which is currently being acquired.

The methods *getReadyFrame...()* relate to the last completely acquired frame.

The methods *getSummedFrames...()* operate to all frames which have been acquired so far. Optional parameter *only_ready_frames* selects if the current incomplete frame shall be included or excluded from calculation.

The methods *get...Ex* instead of an array return a *FlimFrameInfo* object containing frame data with additional information collected at the same time instance.

Parameters

- **tagger** (*TimeTaggerBase*) – time tagger object instance
- **start_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click_channel** (*int*) – channel on which start clicks are received for the time differences histogramming
- **pixel_begin_channel** (*int*) – start marker of a pixel (histogram)
- **n_pixels** (*int*) – number of pixels (histograms) of one frame
- **n_bins** (*int*) – number of histogram bins for each pixel
- **binwidth** (*int*) – bin size in picoseconds
- **pixel_end_channel** (*int*) – end marker of a pixel - incoming clicks on the *click_channel* will be ignored afterward. (optional)
- **frame_begin_channel** (*int*) – start the frame, or reset the pixel index. (optional)
- **finish_after_outputframe** (*int*) – sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0, one frame is stored and the measurement runs continuously. (optional, default: 0)
- **n_frame_average** (*int*) – average multiple input frames into one output frame, (optional, default: 1)
- **pre_initialize** (*bool*) – initializes the measurement on constructing. (optional, default=True). On subclassing, you must set this parameter to False, and then call *.initialize()* at the end of your custom constructor method.

See all common methods

getCurrentFrame()

Returns

The histograms for all pixels of the currently active frame, 2D array with dimensions [n_bins, n_pixels].

Return type

2D_array[int]

getCurrentFrameEx()**Returns**

The currently active frame data with additional information collected at the same instance of time.

Return type*FlimFrameInfo***getCurrentFrameIntensity()****Returns**

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

Return type

1D_array[float]

getFramesAcquired()**Returns**

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

Return type

int

getIndex()**Returns**

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

Return type

1D_array[int]

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

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

Returns

The histograms for all pixels according to the frame index given. If the index is -1, it will return the last frame, which has been completed. When *stop_after_outputframe* is 0, the index value must be -1. If *index* >= *stop_after_outputframe*, it will throw an error. 2D array with dimensions [n_bins, n_pixels]

Return type

2D_array[int]

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

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

Returns

The frame according to the index given. If the index is -1, it will return the latest

completed frame. When *stop_after_outputframe* is 0, index must be -1. If *index* \geq *stop_after_outputframe*, it will throw an error.

Return type

FlimFrameInfo

getReadyFrameIntensity(*[index=-1]*)

Parameters

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

Returns

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

Return type

1D_array[*float*]

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

Parameters

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

Returns

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

Return type

2D_array[*int*]

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

Parameters

- **only_ready_frames** (*bool*) – If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional)
- **clear_summed** (*bool*) – If True, the summed frames memory will be cleared. (optional)

Returns

A sum of all acquired frames with additional information collected at the same instance of time.

Return type

FlimFrameInfo

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

Parameters

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

Returns

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

Return type

1D_array[float]

isAcquiring()

Tells if the class is still acquiring data. It can only reach the false state if `stop_after_outputframe > 0`. This method is different from `isRunning()` and indicates if the specified number of frames is acquired. After acquisition completed, it can't be started again.

Returns

True/False

Return type

bool

frameReady(*frame_number*, *data*, *pixel_begin_times*, *pixel_end_times*, *frame_begin_time*, *frame_end_time*)

Parameters

- **frame_number** (*int*) – current frame number
- **data** (*1D_array[int]*) – 1D array containing the raw histogram data, with the data of pixel *i* and time bin *j* at index $i * n_bins + j$
- **pixel_begin_times** (*list[int]*) – start time for each pixel
- **pixel_end_times** (*list[int]*) – end time for each pixel
- **frame_begin_time** (*int*) – start time of the frame
- **frame_end_time** (*int*) – end time of the frame

The method is called automatically by the Time Tagger engine for each completely acquired frame. In its parameters, it provides FLIM frame data and related information. You have to override this method with your own implementation.

Warning: The code of override must be fast, as it is executed in context of Time Tagger processing thread and blocks the processing pipeline. Slow override code may lead to the buffer overflows.

initialize()

This function initialized the Flim object and starts execution. It does nothing if constructor parameter `pre_initialize==True`.

FlimFrameInfo

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

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

class FlimFrameInfo

pixels: `int`

number of pixels in the frame

bins: `int`

number of bins of each histogram

frame_number: `int`

current frame number

pixel_position: `int`

current pixel position

getFrameNumber()

Returns

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

Return type

`int`

isValid()

Returns

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

Return type

`bool`

getPixelPosition()

Returns

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

Return type

`int`

getHistograms()

Returns

All histograms of the frame, 2D array with dimensions [n_bins, n_pixels].

Return type

`2D_array[int]`

getIntensities()

Returns

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

Return type

`1D_array[float]`

getSummedCounts()

Returns

The summed counts of each histogram.

Return type

`1D_array[int]`

getPixelBegins()**Returns**

An array of the start timestamps of each pixel.

Return type

1D_array[int]

getPixelEnds()**Returns**

An array of the end timestamps of each pixel.

Return type

1D_array[int]

FlimBase

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

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

This is a minimal class that acquires a FLIM frame and calls virtual/abstract *frameReady()* callback method with the frame data as parameters. This class is intended for custom implementations of fast FLIM frame processing with minimal overhead. You can reach frame acquisition rates suitable for realtime video observation.

If you need custom FLIM frame processing implementation while retaining functionality present in the *Flim* class, consider subclassing *Flim* instead.

Warning: When overriding this class, you must set *pre_initialize=False* and then call *initialize()* at the end of your custom constructor code. Otherwise, you may experience unstable or erratic behavior of your program, as the callback *frameReady()* may be called before construction of the subclass completed.

Parameters

- **tagger** (*TimeTaggerBase*) – time tagger object instance
- **start_channel** (*int*) – channel on which clicks are received for the time differences histogramming
- **click_channel** (*int*) – channel on which start clicks are received for the time differences histogramming
- **pixel_begin_channel** (*int*) – start marker of a pixel (histogram)
- **n_pixels** (*int*) – number of pixels (histograms) of one frame
- **n_bins** (*int*) – number of histogram bins for each pixel
- **binwidth** (*int*) – bin size in picoseconds
- **pixel_end_channel** (*int*) – end marker of a pixel - incoming clicks on the click_channel will be ignored afterward. (optional, default: *CHANNEL_UNUSED*)
- **frame_begin_channel** (*int*) – start the frame, or reset the pixel index. (optional, default: *CHANNEL_UNUSED*)

- **finish_after_outputframe** (*int*) – sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0, one frame is stored and the measurement runs continuously. (optional, default: 0)
- **n_frame_average** (*int*) – average multiple input frames into one output frame. (optional, default: 1)
- **pre_initialize** (*bool*) – initializes the measurement on constructing. (optional, default: True). On subclassing, you must set this parameter to False, and then call *.initialize()* at the end of your custom constructor method.

See all common methods

isAcquiring()

Returns

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

Return type

bool

frameReady(*frame_number, data, pixel_begin_times, pixel_end_times, frame_begin_time, frame_end_time*)

Parameters

- **frame_number** (*int*) – current Frame number
- **data** (*1D_array[int]*) – 1D array containing the raw histogram data, with the data of pixel *i* and time bin *j* at index $i * n_bins + j$
- **pixel_begin_times** (*list[int]*) – start time for each pixel
- **pixel_end_times** (*list[int]*) – end time for each pixel
- **frame_begin_time** (*int*) – start time of the frame
- **frame_end_time** (*int*) – end time of the frame

The method is called automatically by the Time Tagger engine for each completely acquired frame. In its parameters, it provides FLIM frame data and related information. You have to override this method with your implementation.

Warning: The code of override must be fast, as it is executed in context of Time Tagger processing thread and blocks the processing pipeline. Slow override code may lead to the buffer overflows.

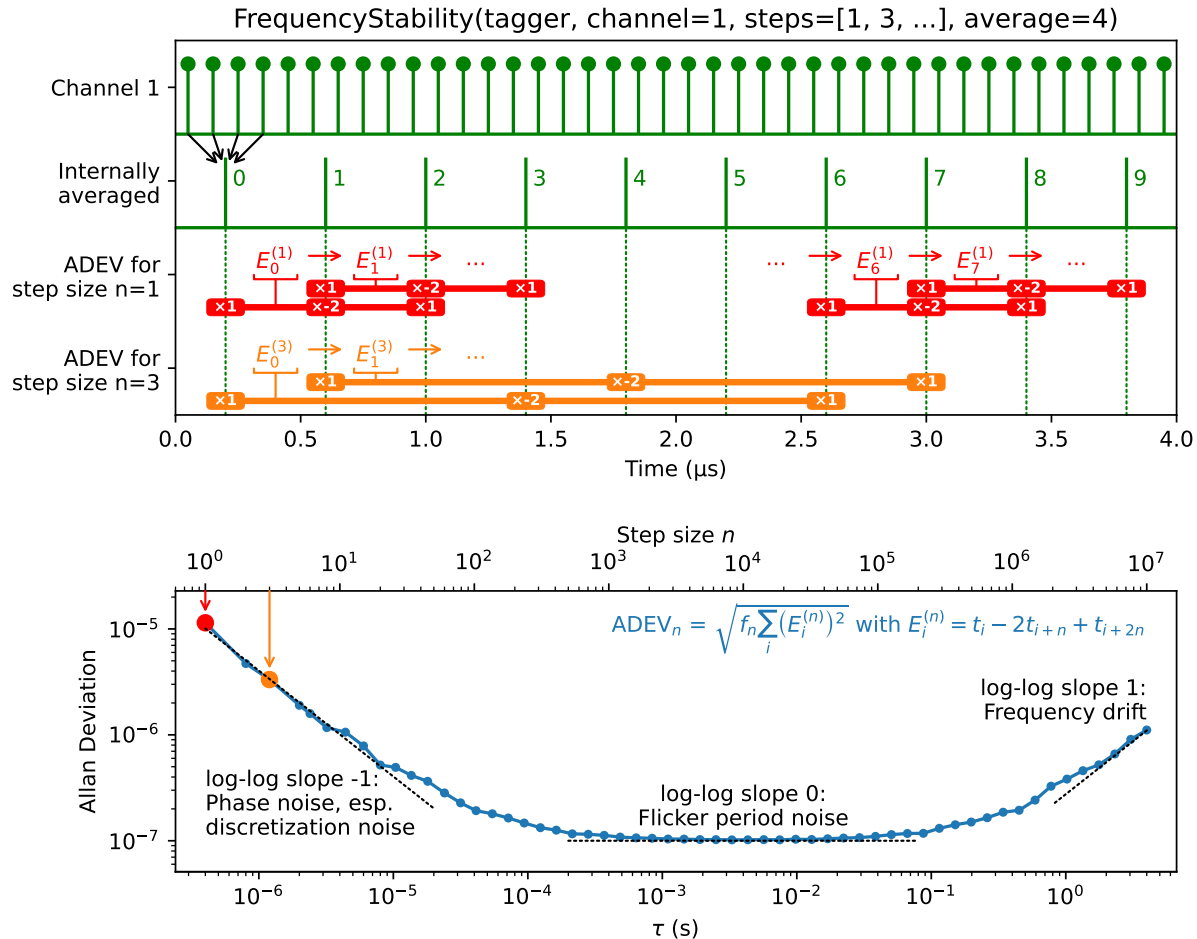
initialize()

This function initialized the Flim object and starts execution. It does nothing if constructor parameter *pre_initialize==True*.

7.5.6 Phase & frequency analysis

This section describes measurements that expect periodic signals, e.g., oscillator outputs.

FrequencyStability



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

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

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

of N averaged timestamps is

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

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

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

class [*FrequencyStability*](#)(tagger, channel, steps, average, trace_len)

Parameters

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

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

See all common methods

getDataObject()

Returns

An object that allows access to the current metrics

Return type

[*FrequencyStabilityData*](#)

class [*FrequencyStabilityData*](#)

getTau()

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

Returns

The τ values.

Return type

1D_array[float]

getADEV()

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

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

Sample

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

Domain

Allan domain

Returns

The overlapping Allan Deviation.

Return type

1D_array[float]

getMDEV()

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

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

The metric is more commonly used in the time domain, see [getTDEV\(\)](#).

Sample

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

Domain

Allan domain

Returns

The overlapping MDEV.

Return type

1D_array[float]

getHDEV()

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

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

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

Sample

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

Domain

Allan domain

Returns

The overlapping Hadamard Deviation.

Return type

1D_array[float]

getSTDD()

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

Standard deviation of the periods.

Sample

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

Domain

Time domain

Returns

The standard deviation.

Return type

1D_array[float]

getADEVScaled()

Domain

Time domain

Returns

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

Return type

1D_array[float]

getTDEV()

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

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

Domain

Time domain

Returns

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

Return type

1D_array[float]

getHDEVScaled()

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

Domain

Time domain

Returns

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

Return type

1D_array[float]

getTraceIndex()

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

Returns

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

Return type

1D_array[float]

getTracePhase()

Provides the time offset of the averaged timestamps from a linear fit over the last `trace_len` averaged timestamps.

Returns

A trace of the last `trace_len` phase samples in seconds.

Return type

1D_array[float]

getTraceFrequency()

Provides the relative frequency offset from the average frequency during the last `trace_len + 1` averaged timestamps.

Returns

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

Return type

1D_array[float]

getTraceFrequencyAbsolute(input_frequency: float = 0.0)

Provides the absolute frequency offset from a given `input_frequency` during the last `trace_len + 1` averaged timestamps.

Parameters

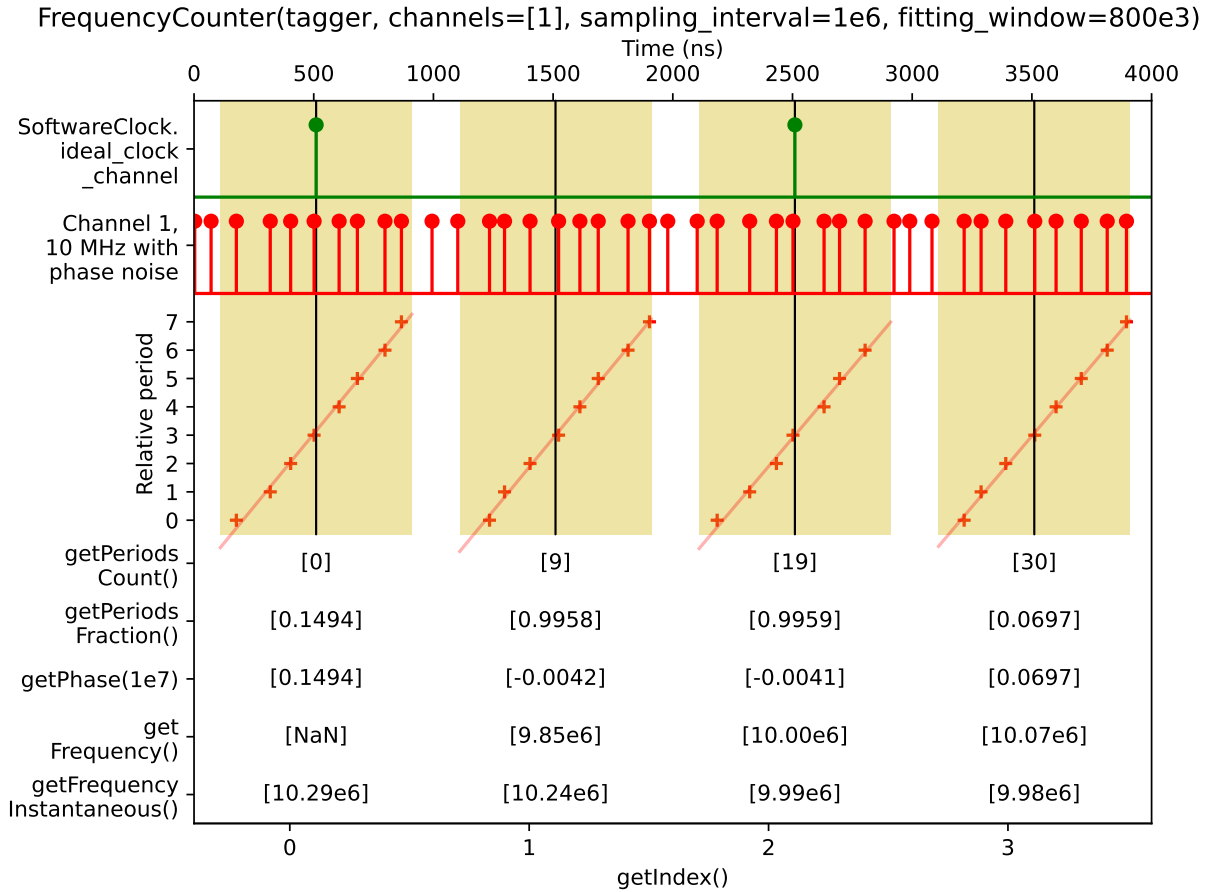
input_frequency (float) – Nominal frequency of the periodic signal. Default is 0 Hz.

Returns

A trace of the last `trace_len` frequency data points in Hz.

Return type
 1D_array[float]

FrequencyCounter



This measurement calculates the phase of a periodic signal at evenly spaced sampling times. If the *SoftwareClock* is active, the sampling times will automatically align with the *SoftwareClockState.ideal_clock_channel*. Multiple channels can be analyzed in parallel to compare the phase evolution in time. Around every sampling time, the time tags within an adjustable *fitting_window* are used to fit the phase.

```
class FrequencyCounter(tagger, channels: list[int], sampling_interval: int, fitting_window: int, n_values: int = 0)
```

Parameters

- **tagger** (*TimeTaggerBase*) – Time Tagger object instance
- **channels** (*list[int]*) – List of channels to analyze
- **sampling_interval** (*int*) – The sampling interval in picoseconds. If the *SoftwareClock* is active, it is recommended to set this value to an integer multiple of the *SoftwareClockState.clock_period*.
- **fitting_window** (*int*) – Time tags within this range around a sampling point are fitted for phase calculation

- **n_values** (*int*) – Maximum number of sampling points to store

See all common methods

getDataObject(*event_divider: int = 1, remove: bool = False*)

Returns a *FrequencyCounterData* object containing a snapshot of the data accumulated in the *FrequencyCounter* at the time this method is called. The *event_divider* argument can be used to scale the results according to the current setting of *TimeTagger.setEventDivider()*. The *remove* argument allows you to control whether the data should be removed from the internal buffer or not.

Parameters

- **event_divider** (*int*) – Compensate for the EventDivider. Default: 1.
- **remove** (*bool*) – Control if data is removed from the internal buffer. Default: *True*.

Returns

An object providing access to a snapshot data.

Return type

FrequencyCounterData

class FrequencyCounterData

size: *int*

Number of sampling points represented by the object.

overflow_samples: *int*

Number of sampling points affected by an overflow range since the start of the measurement.

align_to_reference: *bool*

Indicates if the sampling grid has been aligned to the *SoftwareClock*.

sampling_interval: *int*

The sampling interval in picoseconds.

sample_offset: *int*

Index offset of the first sampling point in the object.

getIndex()

Index of the samples. The reference sample would have index 0, counting starts with 1 at the first sampling point.

Returns

The index of the samples

Return type

1D_array[int]

getTime()

Array of timestamps of the sampling points.

Returns

The timestamps of the sampling points

Return type

1D_array[int]

getPeriodsCount()

The integer part of the phase, i.e. full periods of the oscillation.

Returns

Full cycles per channel and sampling point.

Return type

2D_array[int]

getPeriodsFraction()

The fraction of the current period at the sampling time.

Warning: Be careful with adding `getPeriodsCount()` and `getPeriodsFraction()` as the required precision can overflow a 64bit double precision within minutes. In doubt, please use `getPhase()` with the expected frequency instead.

Returns

A fractional value in range [0, 1) per channel and sampling point.

Return type

2D_array[float]

getPhase(reference_frequency: float = 0.0)

The relative phase with respect to a numerical reference signal, typically at the expected frequency. The reference signal starts at phase 0 at index 0, so the return value of this method is identical to that of `getPeriodsFraction()` for index 0.

Parameters

reference_frequency – The reference frequency in Hz to subtract. Default: 0.0 Hz.

Returns

Relative phase values per channel and sampling point.

Return type

2D_array[float]

getFrequency(time_scale: int = 1_000_000_000_000)

The frequency with respect to the phase difference to the last sampling interval. At index 0, there is no previous phase value to compare with, so the method returns an undefined value *NaN* by definition.

Parameters

time_scale – Scales the return value to this time interval. Default is 1 s, so the return value is in Hz. For negative values, the time scale is set to `sampling_interval`.

Returns

A frequency value per channel and sampling point.

Return type

2D_array[float]

getFrequencyInstantaneous()

The instantaneous frequency with respect to the current fitting window. This value corresponds to the slope of the linear fit.

Returns

An instantaneous frequency value per channel and sampling point.

Return type

2D_array[float]

getOverflowMask()

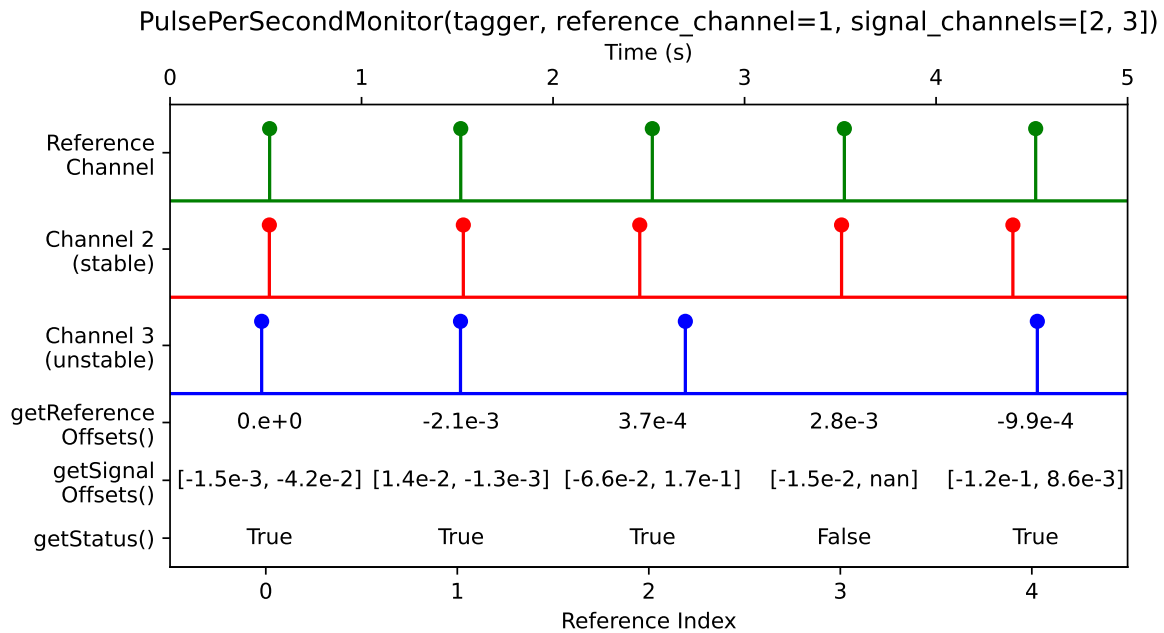
If an overflow range overlaps with a fitting window, the values are invalid. This mask array indicates invalid elements and can be used to filter the results of the other getters.

Returns

1 indicates that the sampling point was affected by an overflow range, 0 indicates valid data.

Return type

2D_array[int]

PulsePerSecondMonitor

Note: *PulsePerSecondMonitor()* and *PulsePerSecondData* are part of the Experimental namespace, and their properties may change in future software versions without further notice. They can be accessed as:

```
TimeTagger.Experimental.PulsePerSecondMonitor(tagger, reference_channel=1, signal_
↪ channels=[2, 3],
                                         filename="output", period=1E12)
```

This measurement allows the user to monitor the synchronicity of different sources of 1 pulse per second (PPS) signals with respect to a reference source. For each signal from the reference PPS source, comparative offsets are calculated for the other signal channels. Upon processing, a UTC timestamp from the system time is associated with each reference pulse.

The monitoring starts on the first signal from the reference source and will run uninterrupted until the measurement is stopped. If a signal from a channel is not detected within one and a half periods, its respective offset will not be calculated but the measurement will continue nonetheless.

By specifying an output file name, the monitoring data can be continuously written to a comma-separated value file (.csv).


```
class PulsePerSecondMonitor(tagger, reference_channel: int, signal_channels: list[int], filename: str = "",
                             period: int = 1E12)
```

Parameters

- **tagger** ([TimeTaggerBase](#)) – Time Tagger object instance.
- **reference_channel** ([int](#)) – the channel number corresponding to the PPS reference source.
- **signal_channels** ([list\[int\]](#)) – a list of channel numbers with PPS signals to be compared to the reference.
- **filename** ([str](#)) – the name of the .csv file to store measurement data. By default, no data is written to file.
- **period** ([int](#)) – the assumed period of the reference source, typically one second, in picoseconds.

See all common methods

```
getDataObject(bool remove = False)
```

Returns a [PulsePerSecondData](#) object containing a snapshot of the data accumulated in the [PulsePerSecondMonitor](#) at the time this method is called. To remove the data from the internal memory after each call, set `remove` to `True`.

Parameters

remove ([bool](#)) – Controls if the returned data shall be removed from the internal buffer.

Returns

An object providing access to a snapshot data.

Return type

[PulsePerSecondData](#)

```
class PulsePerSecondData
```

```
size: int
```

Number of reference pulses contained in the [PulsePerSecondData](#) object.

```
getIndices()
```

The indices of each reference pulse in the [PulsePerSecondData](#) object. The first reference pulse will have index 0, each subsequent pulse from the reference source increments the index by one. In case of overflows in the reference channel, this index will be incremented by the number of missed pulses.

Returns

A list of indices for each pulse from the reference source.

Return type

[1D_array\[int\]](#)

```
getReferenceOffsets()
```

A list of offsets of each reference pulse with respect to its predecessor, with the period subtracted. For a perfect PPS source, this offset would always be zero.

The offset of the first pulse is always defined to be zero. If a reference signal is missing, its offset is defined to be NaN.

Returns

A list of the offsets of each reference with respect to the previous.

Return type

[1D_array\[float\]](#)

getSignalOffsets()

For each reference contained in the *PulsePerSecondData* object a list of offsets for each signal channel is given, in the channel order given by `signal_channels`.

If any signal is missing, its offset is defined to be NaN.

Returns

A list of lists of offsets for each `signal_channel` for given reference pulses.

Return type

2D_array[float]

getUtcSeconds()

The number of elapsed seconds from the beginning of the Unix epoch (1st of January 1970) to the time at which each reference pulse is processed, as a floating point number.

Returns

A list of the number of seconds since the Unix epoch to the time of processing, for each reference pulse.

Return type

1D_array[float]

getUtcDates()

The UTC timestamps for the system time at which each reference pulse is processed, as a string with ISO 8601 formatting (YYYY-MM-DD hh:mm:ss.ssssss).

Returns

A list of the UTC timestamp at processing time, for each reference pulse.

Return type

1D_array[str]

getStatus()

A list of booleans values describing whether all signals, including from the reference source, were detected. True corresponds to a complete collection of signals, False otherwise.

Returns

A list of bools describing the signal integrity for each reference pulse.

Return type

1D_array[bool]

7.5.7 Time-tag-streaming

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

Time tag format

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

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

TimeTagStream

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

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

Parameters

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

See all common methods

getData()

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

Returns

buffer object containing timetags collected.

Return type

TimeTagStreamBuffer

```
class TimeTagStreamBuffer
```

size: *int*

Number of events stored in the buffer. If the size equals the maximum size of the buffer set in *TimeTagStream* via *n_max_events*, events have likely been discarded.

hasOverflows: *bool*

Returns True if a stream overflow was detected in any of the tags received. Note: this is independent of an overflow of the internal buffer of *TimeTagStream*.

tStart: *int*

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

tGetData: *int*

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

getTimestamps()

Returns an array of timestamps.

Returns

Event timestamps in picoseconds for all chosen channels.

Return type

`list[int]`

getChannels()

Returns an array of channel numbers for every timestamp.

Returns

Channel number for each detected event.

Return type

`list[int]`

getOverflows()

Deprecated since version 2.5.0: please use `getEventTypes()` instead.

getEventTypes()

Returns an array of event type for every timestamp. See, *Time tag format*. The method returns plain integers, but you can use *TagType* to compare the values.

Returns

Event type value for each detected event.

Return type

`1D_array[int]`

getMissedEvents()

Returns an array of missed event counts during an stream overflow situation.

Returns

Missed events value for each detected event.

Return type

`1D_array[int]`

FileWriter

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

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

The *FileWriter* is able to split the data into multiple files seamlessly when the file size reaches a maximal size. For the file splitting to work properly, the filename specified by the user will be extended with a suffix containing sequential counter, so the filenames will look like in the following example

```
fw = FileWriter(tagger, 'filename.ttbin', [1,2,3]) # Store tags from channels 1,2,3

# When splitting occurs the files with following names will be created
# filename.ttbin      # the sequence header file with no data blocks
# filename.1.ttbin    # the first file with data blocks
# filename.2.ttbin
# filename.3.ttbin
# ...
```

In addition, the *FileWriter* will query and store the configuration of the Time Tagger in the same format as returned by the *TimeTaggerBase.getConfiguration()* method. The configuration is always written into every file.

See also: *FileReader*, *The TimeTaggerVirtual class*, and *mergeStreamFiles()*.

class *FileWriter*(tagger, filename, channels)

Parameters

- **tagger** (*TimeTaggerBase*) – time tagger object
- **filename** (*str*) – name of the file to store to
- **channels** (*list[int]*) – non-empty list of real or virtual channels

Class constructor. As with all other measurements, the data recording starts immediately after the class instantiation unless you initialize the *FileWriter* with a *SynchronizedMeasurements*.

Note: Compared to the *Dump* measurement, the *FileWriter* requires explicit specification of the channels. If you want to store timetags from all input channels, you can query the list of all input channels with *TimeTagger.getChannelList()*.

See all common methods

split([*new_filename=""*])

Close the current file and create a new one. If the *new_filename* is provided, the data writing will continue into the file with the new filename and the sequence counter will be reset to zero.

You can force the file splitting when you call this method without parameter or when the *new_filename* is an empty string.

Parameters

new_filename (*str*) – filename of the new file.

setMaxFileSize(*max_file_size*)

Set the maximum file size on disk. When this size is exceeded a new file will be automatically created to continue recording. The actual file size might be larger by one block. (default: ~1 GByte)

getMaxFileSize()

Returns

The maximal file size. See also *FileWriter.setMaxFileSize()*.

Return type

int

getTotalEvents()

Returns

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

Return type

int

getTotalSize()**Returns**

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

Return type

int

FileReader

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

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

Example:

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

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

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

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

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

See also: *FileWriter*, *The TimeTaggerVirtual class*, and *mergeStreamFiles()*.

class FileReader(filenames)

This is the class constructor. The *FileReader* automatically continues to read files that were split by the *FileWriter*.

Parameters

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

getData(size_t n_events)

Reads the next *n_events* and returns the buffer object with the specified number of timetags. The *FileReader* stores the current location in the data file and guarantees that every timetag is returned once. If less than *n_elements* are returned, the reader has reached the end of the last file in the file-list *filenames*. To check if more data is available for reading, it is more convenient to use *hasData()*.

Parameters

n_events (*int*) – Number of timetags to read from the file.

Returns

A buffer of size *n_events*.

Return type

TimeTagStreamBuffer

hasData()**Returns**

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

Return type

bool

getConfiguration()**Returns**

A JSON formatted string (*dict* in Python) that contains the Time Tagger configuration at the time of file creation.

Return type

str or *dict*

getChannelList()**Returns**

all channels available within the input file

Return type

list[int]

Dump

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

Warning: The files created with this class are not readable by *TimeTaggerVirtual* and *FileReader*.

Writes the timetag stream into a file in a simple uncompressed binary format that store timetags as 128bit records, see *Time tag format*.

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

class Dump(*tagger, filename, max_tags, channels*)

Parameters

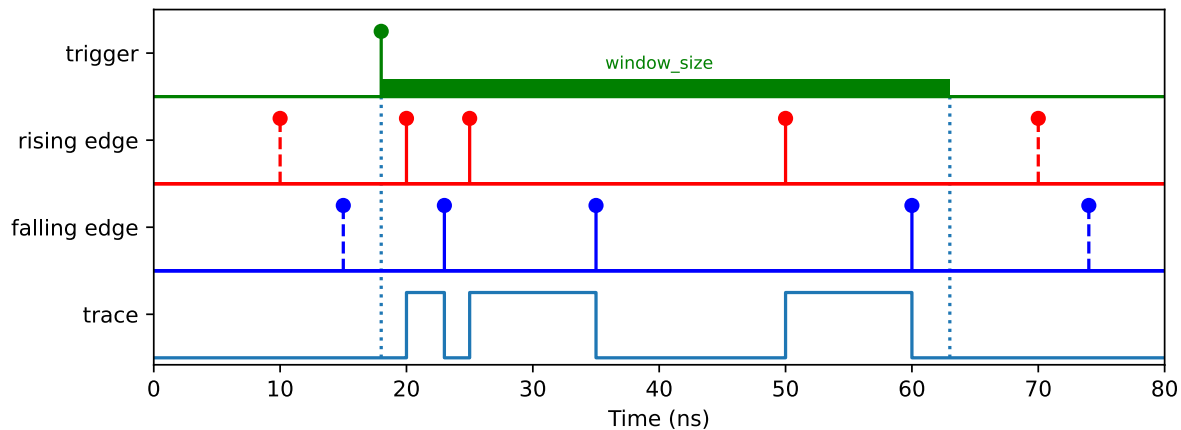
- **tagger** (*TimeTaggerBase*) – time tagger object instance
- **filename** (*str*) – name of the file to dump to
- **max_tags** (*int*) – stop after this number of tags has been dumped. Negative values will dump forever
- **channels** (*list[int]*) – list of real or virtual channels which are dumped to the file (when empty or not passed all active channels are dumped)

clear()

Delete current data in the file and restart data storage.

stop()

Stops data recording and closes data file.

Scope

The [Scope](#) class allows to visualize time tags for rising and falling edges in a time trace diagram similarly to an ultrafast logic analyzer. The trace recording is synchronized to a trigger signal which can be any physical or virtual channel. However, only physical channels can be specified to the `event_channels` parameter. Additionally, one has to specify the time `window_size` which is the timetrace duration to be recorded, the number of traces to be recorded and the maximum number of events to be detected. If `n_traces < 1` then retriggering will occur infinitely, which is similar to the “normal” mode of an oscilloscope.

Note: Scope class implicitly enables the detection of positive and negative edges for every physical channel specified in `event_channels`. This accordingly doubles the data rate requirement per input.

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

Parameters

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

See all common methods

getData()

Returns a tuple of the size equal to the number of `event_channels` multiplied by `n_traces`, where each element is a tuple of [Event](#).

Returns

Event list for each trace.

Return type

`tuple[tuple[Event]]`

ready()**Returns**

Returns whether the acquisition is complete which means that all traces (*n_traces*) are acquired.

Return type

`bool`

triggered()**Returns**

Returns number of trigger events have been captured so far.

Return type

`int`

class Event

Pair of the timestamp and the new state.

time**Type**

`int`

state**Type**

State

class State

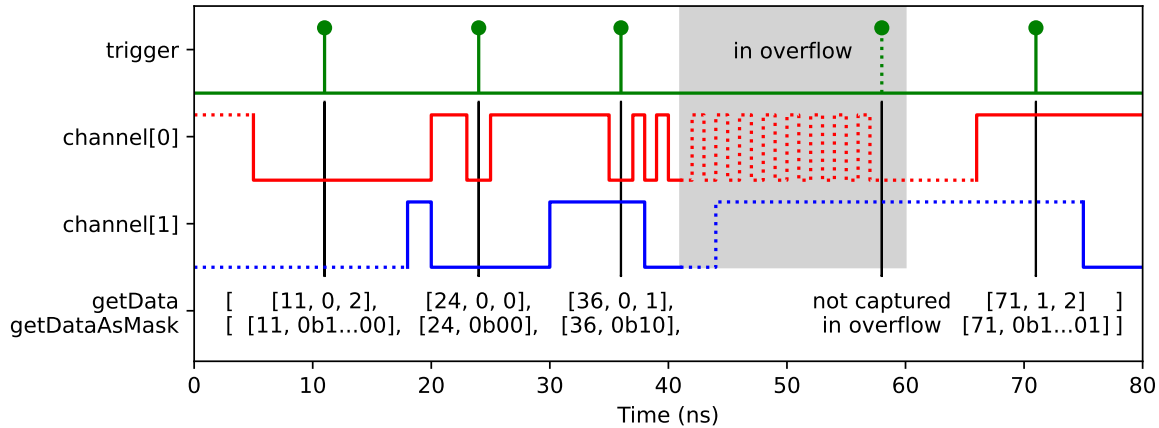
Current input state. Can be unknown because no edge has been detected on the given channel after initialization or an overflow.

UNKNOWN

HIGH

LOW

Sampler



The *[Sampler](#)* class allows sampling the state of a set of channels via a trigger channel.

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

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

The maximum number of channels is limited to 63 for one *[Sampler](#)* instance.

class *[Sampler](#)*(tagger, trigger, channels, max_trigger)

Parameters

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

See all common methods

getData()

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

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

Where state means:

```

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

```

Returns

sampled data

Return type

2D_array[int]

getDataAsMask()

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

```

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

```

Where state means:

```

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

```

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

Returns

sampled data

Return type

2D_array[int]

7.5.8 Helper classes

SynchronizedMeasurements

The *SynchronizedMeasurements* class allows for synchronizing multiple measurement classes in a way that ensures all these measurements to start, stop simultaneously and operate on exactly the same time tags. You can pass a Time Tagger proxy-object returned by *SynchronizedMeasurements.getTagger()* to every measurement you create. This will simultaneously disable their autostart and register for synchronization.

class SynchronizedMeasurements(tagger)

Parameters**tagger** (TimeTaggerBase) – TimeTagger object**registerMeasurement**(measurement)Registers the *measurement* object into a pool of the synchronized measurements.

Note: Registration of the measurement classes with this method does not synchronize them. In order to start/stop/clear these measurements synchronously, call these functions on the

SynchronizedMeasurements object after registering the measurement objects, which should be synchronized.

Parameters

measurement – Any measurement (*IteratorBase*) object.

unregisterMeasurement(*measurement*)

Unregisters the *measurement* object out of the pool of the synchronized measurements.

Note: This method does nothing if the provided measurement is not currently registered.

Parameters

measurement – Any measurement (*IteratorBase*) object.

start()

Calls *start()* for every registered measurement in a synchronized way.

startFor(*duration*[, *clear=True*])

Calls *startFor()* for every registered measurement in a synchronized way.

stop()

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

clear()

Calls *clear()* for every registered measurement in a synchronized way.

isRunning()

Calls *isRunning()* for every registered measurement and returns true if any measurement is running.

getTagger()

Returns a proxy tagger object which can be passed to the constructor of a measurement class to register the measurements at initialization to the synchronized measurement object. Those measurements will not start automatically.

Note: The proxy tagger object returned by *getTagger()* is not identical with the *TimeTagger* object created by *createTimeTagger()*. You can create synchronized measurements with the proxy object the following way:

```
tagger = TimeTagger.createTimeTagger()
syncMeas = TimeTagger.SynchronizedMeasurements(tagger)
taggerSync = syncMeas.getTagger()
counter = TimeTagger.Counter(taggerSync, [1, 2])
countrate = TimeTagger.Countrate(taggerSync, [3, 4])
```

Passing *tagger* as a constructor parameter would lead to the not synchronized behavior.

7.5.9 Custom Measurements

The class `CustomMeasurement` allows you to access the raw time tag stream with very little overhead. By inheriting from `CustomMeasurement`, you can implement your fully customized measurement class. The `CustomMeasurement.process()` method of this class will be invoked as soon as new data is available.

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

```
class CustomMeasurement(tagger)
```

Parameters

tagger (`TimeTaggerBase`) – TimeTagger object

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

See all common methods

```
process(incoming_tags, begin_time, end_time)
```

Parameters

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

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

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

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

mutex

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

```
class MyMeasurement(CustomMeasurement):

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

(continues on next page)

(continued from previous page)

```
with self.mutex:  
    return self.data.copy()
```

IN DEPTH GUIDES

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

8.1 Conditional Filter

The Conditional Filter is a hardware feature that allows you to remove irrelevant time tags carrying no information. In a typical use case, you have a high-frequency signal applied to at least one channel. Examples include fluorescence lifetime measurements or optical quantum information and cryptography where you want to capture synchronization clicks from a high repetition rate excitation laser.

The Conditional Filter distinguishes between *trigger* channels and *filtered* channels. All input channels of your Time Tagger are fully equivalent and can be used as both, trigger or filtered channels. The data rate of the filtered channels will be reduced. The reduction is controlled by the trigger channels: Every trigger opens the gate for an event of the filtered channel. All other events in the filtered channels will be discarded on the Time Tagger and do not need to be transferred via the USB connection.

Being a hardware feature, the Conditional Filter is not controlled on the level of individual measurements. It is enabled on the level of your physical device with a typical Python code looking like

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

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

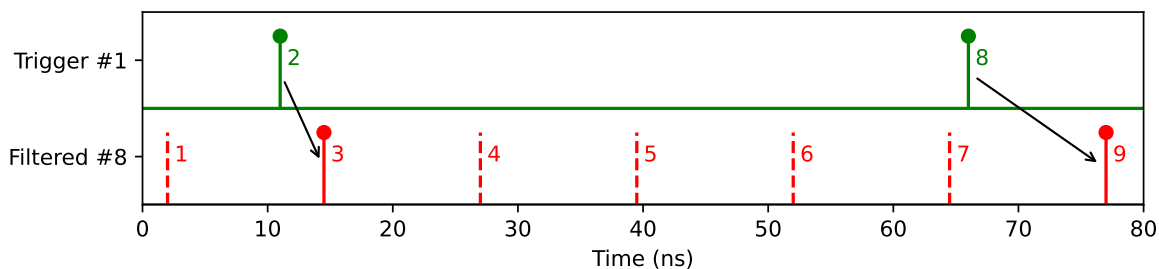
8.1.1 Example configurations

One trigger and one filtered channel

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

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

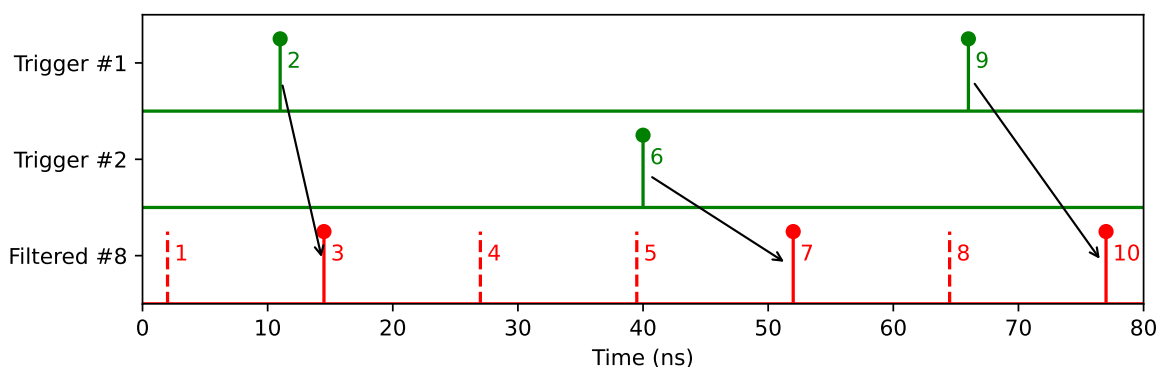
The Conditional Filter discards by default all signals of the filtered-channel. Only the very next event is transmitted after an event on the trigger-channel. In the example, click 2 opens the gate for click 3. When click 3 passes, it closes the gate and the subsequent events will be discarded until another event (click 8) occurs in the trigger channel.



Multiple trigger-channels

There is the option to define more than one trigger-channel for the Conditional Filter. As a consequence, the next event on the filtered-channel is transmitted when there was a event at *any* of the trigger-channels:

```
tagger.setConditionalFilter(trigger=[1, 2], filtered=[8])
```

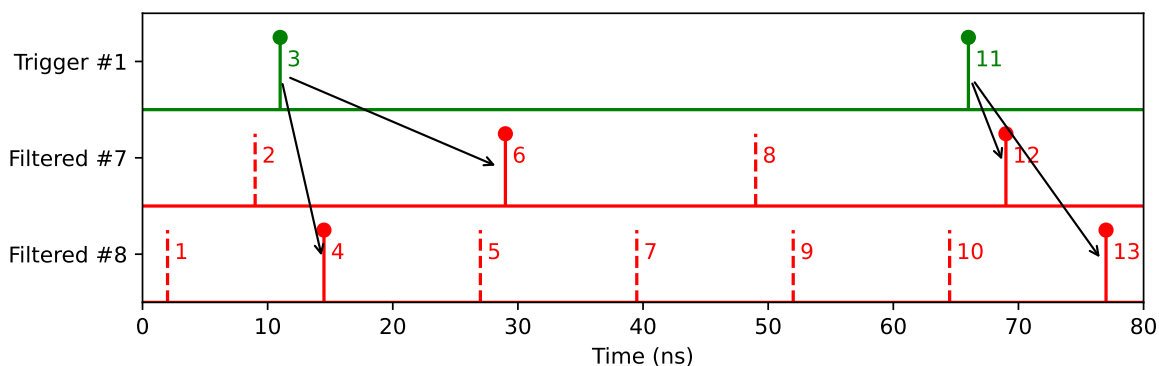


This is the typical use case when you detect photons with multiple detectors and want to correlate both with the common excitation laser.

Multiple filtered channels

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

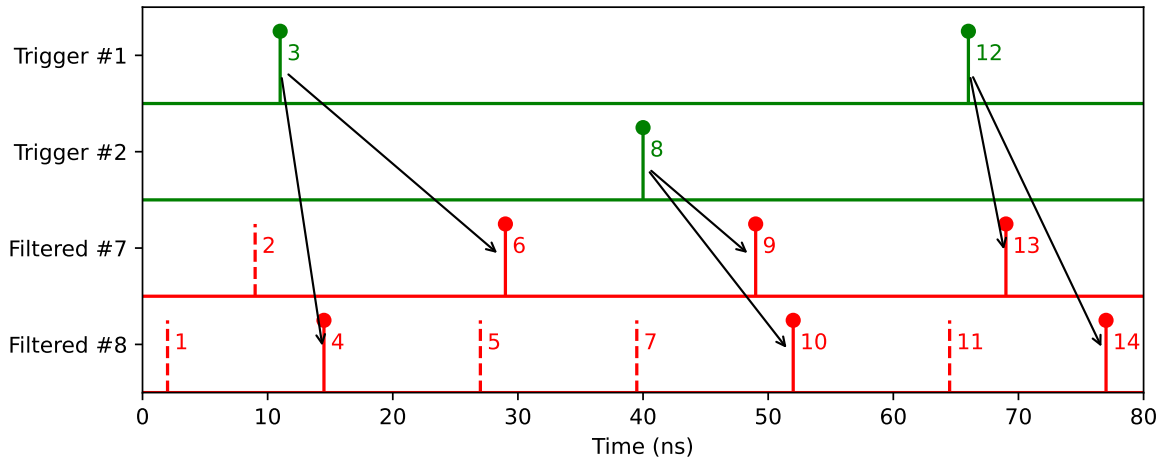
```
tagger.setConditionalFilter(trigger=[1], filtered=[7, 8])
```



Multiple trigger and filtered channels

In general, you can also combine multiple trigger-channels and multiple filtered-channels:

```
tagger.setConditionalFilter(trigger=[1, 2], filtered=[7, 8])
```



This scheme shows two different high-frequency signals on channels #7 and #8. Such cases can occur when you want to run two completely independent experiments on a single Time Tagger. For instance, channels #1/#7 and #2/#8 may represent the two experiments. It is not possible to set up two independent Conditional Filters for these groups. The scheme shown is the only way to apply the Conditional Filter in this case - with the drawback that channel #1 (#2) may also trigger channel #8 (#7), making the filtering less efficient.

8.1.2 Understanding the filtering mechanism

The Conditional Filter is a hardware feature that is embedded in a sequence of processing stages. It is important to understand the order of these stages. Some unexpected results can occur when you are not aware of these mechanisms, so read the following section with care.

Terms

Input time stamp

This is the time stamp *you* are interested in: It refers to the time when the input signal transits the trigger level at the input connector.

TDC time stamp

This is the time stamp *the Time Tagger* is interested in: It is the raw 64 bit integer the FPGA attributes to a pulse edge.

Hardware delay

The signal entering the input connector is routed through the Time Tagger into the FPGA where the time to digital conversion is performed. This route differs from channel to channel and so does the accumulated delay. Because of this, we need to distinguish between *Input time stamp* and *TDC time stamp*. The *hardware delay* cannot be controlled by the user, it is defined by the design of the Time Tagger hardware and the FPGA configuration (this can vary from software release to software release). But don't worry, the Time Tagger is calibrated to compensate for this delay. This compensation is done on the device in case of the *Time Tagger Ultra* and the *Time Tagger X*. The *Time Tagger 20* can only apply the delay in software (see details below). Except for the purpose of understanding the Conditional Filter, you do not need to care about the difference.

External delay

Any delay introduced before the Time Tagger, e.g. by cable lengths or optical pathways.

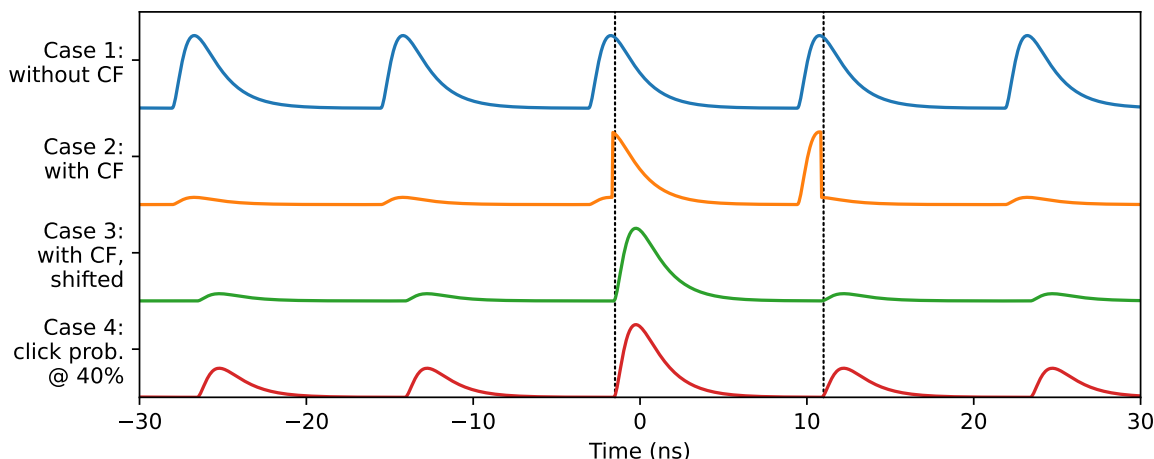
Processing stages

1. **Pulse enters the Time Tagger:** Up to the input connector, the user is in charge of the *external delays*. They can be controlled by changing cable lengths or optical pathways. The time tag generated by the Time Tagger should therefore represent the temporal order at the input connectors. This is the *input time stamp*.
2. **Time to digital conversion:** The pulses propagate through the Time Tagger. They are compared to the trigger level of the input stage. This results in a high or low logic level. This is still analog information that propagates to the FPGA. Here, the *TDC time stamp* is attributed to the pulse edge. The propagation length up to this time to digital conversion (TDC) differs from channel to channel. It can be compensated in one of the later stages.
3. **Adjustable hardware delay (TT Ultra and TTX only!):** From software version 2.8.0 on, the *Time Tagger Ultra* is able to buffer and reorder the tags before the Conditional Filter. For *Time Tagger X*, this feature is available from software version 2.12.0 on. You can set an individual delay for every input stage by *TimeTaggerBase.setDelayHardware()*. This behaves like an adjustable hardware delay and is calibrated by default to compensate for the physical hardware delay. It changes the behavior of the Conditional Filter tremendously, as you will see in the next stages.
4. **Adjustable deadtime:** As a first filter stage, the adjustable deadtime is applied. It acts only on the channel itself, considering rising and falling edges as two separate channels. After an event in one of the channels occurred, no other event can appear in the same channel for the defined deadtime. On the *Time Tagger 20*, the deadtime can only be set in integer multiples of the FPGA clock cycle of 6 ns with a technically required minimum of one cycle. Conversely, on the *Time Tagger Ultra* and *Time Tagger X*, the deadtime can be set to any integer value greater than the duration of one FPGA clock cycle, which is 2000 ps and 1333 ps, respectively.
5. **Conditional Filter:** As a second filter stage, the Conditional Filter is applied. The time tags of trigger channels and filtered channels are compared. If your device is able to introduce *Adjustable hardware delay*, this happens based on the timestamp including the *Hardware delay* compensation and the additional delay set by *TimeTaggerBase.setDelayHardware()*. Otherwise, the raw *TDC time stamp* is used. In both cases, the time order of these stamps can deviate from the order of the *input time stamps* that you are dealing with usually. Note: In the edge case of events arriving at the same time ($dt=0$) on a trigger and filtered channel, it is not specified whether the event on the filtered channel at $dt=0$ is passed through, or the subsequent, or both.
6. **Event Divider:** As a third filter stage, the Event Divider can be applied. Only every n -th time tag of the respective channel is transmitted, all others are dismissed.
7. **The bottleneck - USB transfer:** The time tags are buffered and transmitted to the PC. At this point, after applying Conditional Filter and Event Divider, it is important that the resulting data rate on average does not exceed the maximum data rate.
8. **setDelaySoftware:** From now on, the Time Tagger hardware is not involved anymore. If your device does not provide an adjustable hardware delay, the software compensates now the *TDC time stamp* for the *hardware delay* to provide you the *input time stamp* (it is possible to disable the hardware delay compensation, see *Control hardware delay compensation*). In any case, you can modify this compensation by *TimeTaggerBase.setDelaySoftware()*.
9. **Delayed Channel:** The most flexible way to control the relative delay of your signals are Virtual Channels.

Consequences

The nature of the filtering process can produce counterintuitive results that need to be handled. We will explore these cases based on the example of a fluorescence lifetime measurement. The sample is excited by a pulsed laser with a repetition rate of 80 MHz (period of 12.5 ns), the laser synchronization signal is connected to channel #8. So channel #8 is the high-frequency input that needs to be filtered. Fluorescence photons are collected by a single-photon detector connected to channel #1 that will trigger the Conditional Filter. We set up a correlation measurement and look at different cases:

```
TimeTagger.Correlation(tagger, 1, 8)
```

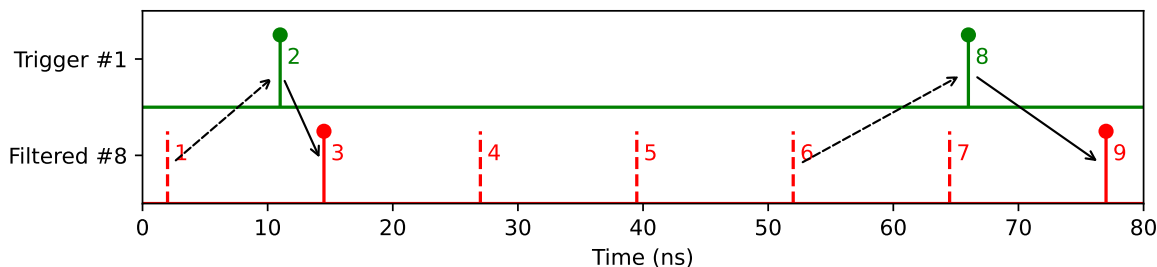


Case 1: Without the Conditional Filter set up, the Correlation measurement class provides a periodic signal. The periodicity is a result of the multi-start/multi-stop approach of the Correlation measurement: A click on the detector will contribute together with any laser synchronization pulse to the correlation, not only with the one that actually stimulated the photon. Without the Conditional Filter, there will be a laser time tag every 12.5 ns. Because this high frequency cannot be transferred for a long time, buffer overflows will lead to discarded data.

Case 2: With the Conditional Filter on, the data rate is highly reduced at the cost of losing the full periodicity of the signal:

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

Now we observe that the majority of the events is in the range of a few nanoseconds. However, the signal does not look like expected: Instead of a signal resembling one of the peaks from Case 1, a double peak appears. If you look carefully at the signal, you can see that the lifetime curve is cut along the dotted line and one part is shifted by one period. This indicates that the physical delay between the input channels is not designed properly. The scheme illustrates the problem:



The dashed line indicates which pulse excited the sample. If the photon is emitted early by the sample (click 2), it will

trigger the first pulse (click 3) after the stimulating one (click 1). In the second case, the photon is emitted late and the subsequent laser pulse (click 7) has already passed. In this case, click 9 is passed and click 8 seems to be very early, although it is quite late, in fact.

Case 3: To align the signal properly, having the signal in between two laser events, the strategy depends on your device: With Time Tagger Ultra (with software version 2.8.0 and later) and Time Tagger X (with software version 2.12.0 and later), you can use `TimeTaggerBase.setDelayHardware()` to align your signals. In the case of a Time Tagger 20, however, you need to adjust your *external delays*. You might either modify optical path lengths or use cables of different lengths.

Case 4: This case illustrates that the height of the higher-order peaks is determined by the count rate of your detector. The relative height (compared to the center peak) is proportional to the probability for a laser synchronization pulse to pass the Conditional Filter in the higher-order period. This probability is given by the probability that a detector click occurs in the respective period and gates the synchronization click. In Case 1, without the Conditional Filter, the probability is 100% - every synchronization pulse is passed. For Case 2 and Case 3, the probability has been set to 10%, in Case 4 it has been increased to 40%.

Note: In Cases 3 and 4, with *external delays* well adjusted to each other, you can see a signal at negative times. How is this possible? Wouldn't this mean that the laser synchronization click arrived earlier than the photon click that gated it? Does my Time Tagger violate causality?

The answer is: No, it does not. The occurrence of negative delays is caused by the difference between the *input time stamps* and the *TDC time stamps*. Negative delays occur in *input time stamps*, but causality must only be obeyed in *TDC time stamps* (plus *adjustable onboard delays*, if available). The occurrence of negative delays indicates that the *hardware delay* of channel #8 (laser synchronization) is larger than that of channel #1 (detector).

8.1.3 Setup of the Conditional Filter

The `TimeTagger.setConditionalFilter()` method expects two arguments, *trigger* and *filtered*, and accepts the optional boolean argument *hardwareDelayCompensation*:

```
tagger.setConditionalFilter(trigger: list[int],
                           filtered: list[int],
                           hardwareDelayCompensation: bool = True)
```

The effect of *trigger* and *filter* can be reviewed in the *Example configurations* section.

Control hardware delay compensation

With the argument *hardwareDelayCompensation* you can decide whether the *hardware delay* is compensated or not. This means, in fact, that you can decide whether you work with *input time stamps* or with *TDC time stamps*. If your device supports *adjustable onboard delays*, you should never set this value to False and you can ignore this section.

hardwareDelayCompensation = True (default)

Pros

- Time tags are provided in the way you are used to it
- The signal position will not depend on the software version

Cons

- Negative time differences can occur between trigger-channel and filtered-channel and seemingly violate causality

hardwareDelayCompensation = False**Pros**

- Provided Time tags will be in the same temporal order as for the ConditionalFilter, no negative time differences will occur

Cons

- Signal positions may change upon software update
- Affects all channels, not only the ones listed in *trigger* and *filtered*.

Disable the Conditional Filter

To disable the Conditional Filter, you can either pass an empty lists or use the `TimeTagger.clearConditionalFilter()` method:

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

8.2 Raw Time-Tag-Stream access

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

8.2.1 Dumping and post-processing

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

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

File Reader

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

Virtual Time Tagger

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

8.2.2 On-the-fly processing

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

TimeTagStream - high-level, lower performance

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

CustomMeasurement - low-level, higher performance

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

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

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

IteratorBase - C++ only

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

8.3 Synchronization of the Time Tagger pipeline

In order to achieve a real-time evaluation of the events with high data rates, the Time Tagger series uses a pipeline based parallel processing.

The hardware records a timestamp for every incoming event and stores it in a large on-device buffer. The size of this buffer can be configured with *setHardwareBufferSize()*. The buffer contents are read by computer over USB, typically in blocks of 128k events or when the time between the blocks exceeds 20 ms. Waiting until a block of data is available is aimed at optimizing the USB throughput while limiting the time between consecutive block allows for reducing data latency on slow event rates. The block size can be tuned by a user with *setStreamBlockSize()*. On the computer, the blocks of data are processed by all running measurements in the order in which the measurements were created. Only one measurement has access to a block at any given time. Once a measurement has finished processing the block, it is ready to process the next block while the previous block becomes available to the next measurement.

Naturally, the transferring and processing of the data takes time and results in the latency. The latency between signal arrival and its appearance in the measurement data is usually below 100 ms; however, it can become as large as a few seconds if the on-device buffer fills up faster than the computer can transfer and process the data.

Proper operation of the pipeline and the control of the device parameters requires a suitable synchronization method. Time Tagger uses the concept of fencing. A fence is a unique identifier that is sent by the software to the hardware. It is added at the end of the on-device buffer data, streamed back to the computer along with timestamp data, and processed by all measurement classes. Once the Time Tagger software detects the fence, it knows that it is located at the data position which was in the buffer when the fence was created. The usefulness of fencing is easily demonstrated with a

following example. When you create a measurement, you expect that it starts processing data from that very instance of time; however, it starts processing the data, which was recorded earlier and is already available in the buffer. With fencing, the measurement creates a fence and begins data accumulation only when it receives the fence back. In this way, the measurement is dealing with the data recorded as close to the measurement creation as possible and avoids processing of the older data.

You can use the fencing mechanism manually. First, you have to create a new fence with `TimeTaggerBase.getFence()` and then wait for it to be signaled with `waitForFence()` at any time later. If you want to create a fence and immediately wait for it then using the `sync()` method is more convenient.

8.4 FPGA link

Warning: The reference design and the on-the-wire format are not stable and will be subject to incompatible changes with further development.

The FPGA link output of the *Time Tagger X* allows you to connect an FPGA of your own design to the *Time Tagger X* via an Ethernet-based protocol and benefit from higher data throughput and lower latency compared to USB.

In a typical use case, you want to use either process tags at a higher rate than the USB connection to the PC allows or integrate the measurements into a test fixture and trigger events based on the measurements.

The SFP+ port on the *Time Tagger X* can be used either with a DAC or fiber transceivers to connect to your own FPGA. We recommend using the [OpalKelly XEM8320](#) for your custom design.

The QSFP+ port on the *Time Tagger X* should be used with a fiber transceiver to connect to your own FPGA. We also recommend using the [OpalKelly XEM8320](#) together with the [SZG-QSFP](#) for your custom design.

Note: We recommend using the SFP+ port unless the higher bandwidth is necessary.

Warning: There is currently no retransmission support so if corruption occurs during transmission, tags will be permanently lost. Please verify that your data link is of high quality or that tag loss can be tolerated.

8.4.1 Getting Started with SFP+

To enable the FPGA link output of the Time Tagger use `enableFpgaLink()`. Start by enabling the FPGA link on channel 1:

```
import TimeTagger
tagger = TimeTagger.createTimeTagger()
tagger.enableFpgaLink([1], "", TimeTagger.FpgaLinkInterface.SFPP_10GE, True)
```

To receive the tags, use our [Time Tagger FPGA link reference](#) design. Follow the instructions in the [XEM8320 readme](#) to build the reference design. Connect the SFP+ port on the *Time Tagger X* to the SFP 1 port on the XEM8320 and load the bitstream on the XEM8320. You should now be able to observe the LED D1 on the XEM8320 matching the input state on channel 1 of the *Time Tagger X*.

To verify the link quality, activate a test signal as follows:

```
tagger.setTestSignal([1], True)
```

and reload the bitstream on the XEM8320. LED D6 should stay dark, indicating that the channel 1 events are arriving at the expected time without drops.

8.4.2 Using QSFP+

QSFP+ is quite similar to using SFP+. Start by enabling the FPGA link for the QSFP+ interface for input channel 1 of the *Time Tagger X*:

```
import TimeTagger
tagger = TimeTagger.createTimeTagger()
tagger.enableFpgaLink([1], "", TimeTagger.FpgaLinkInterface.QSFPP_40GE, True)
```

Similarly use our [Time Tagger FPGA link reference design](https://github.com/swabianinstruments/TimeTagger-FPGA-Link-Reference/tree/main/target/opalkelly-xem8320-qsfp), but follow the instructions in the [40G readme](https://github.com/swabianinstruments/TimeTagger-FPGA-Link-Reference/tree/main/target/opalkelly-xem8320-qsfp).

Connect the QSFP+ port of the *Time Tagger X* with the SZG-QSFP module which has to be connected to Port E of the XEM8320. You should now be able to observe the LED D1 on the XEM8320 matching the input state on channel 1 of the *Time Tagger X*.

Note: Using the reference design with QSFP+ requires the Xilinx *EF-DI-LAUI-SITE* IP core license. We recommend starting with the SFP+ connection.

Warning: Only one output is active at the same time. Enabling the QSFP+ port disables the SFP+ port and vice-versa.

8.4.3 Modifying the reference design

Follow the instructions in [Building you own design](#).

USAGE STATISTICS COLLECTION

You can help us developing and improving the Time Tagger by enabling automated usage statistics collection. The usage statistics data collection is designed to help us better understand how the Time Tagger hardware and software are used. This data includes the performance indicators, configuration, the state of the Time Tagger, and API usage patterns. The usage statistics data is pseudonymized¹ and cannot be linked to a specific user or specific hardware unit. On installation of the Time Tagger software, a random *user_id* will be created and added to the usage statistics reports. Users can review the contents of usage statistics data by using the `getUsageStatisticsReport()`. Also users can disable usage statistics data collection at any time via Time Tagger API as `setUsageStatisticsStatus(UsageStatisticsStatus.Disabled)`. It is possible to enable the usage statistics collection temporarily and without automatic uploading which may be helpful for debugging.

9.1 Contents of the usage statistics data

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

9.2 Ways of control

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

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

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

Get and set usage statistics collection status

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

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

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

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

Get current usage statistics data

```
json_string = getUsageStatisticsReport()
```

9.3 Time Tagger Lab diagnostics




When Time Tagger Lab is used, it can automatically send application error reports and related diagnostics (later: “application error diagnostics”), as described in the EULA. Users can enable/disable sending application error diagnostics and sending Time Tagger usage statistics independently. By default sending application error diagnostics is enabled when sending usage data is enabled.

LEGAL & SAFETY

10.1 *Time Tagger X* - Safety notice

Swabian Instruments's *Time Tagger X* is a high-resolution streaming time-to-digital converter with a timing resolution of 1.5 ps and up to 18 input channels. It comes bundled with a unique data processing architecture that makes it the preferred choice for applications like Time-Correlated Single-Photon Counting (TCSPC), time-interval counting and many more.

10.1.1 Symbols

	Caution: general warning
	Caution: high voltage
	Functional earth

10.1.2 Operation environment

The *Time Tagger X* is designed for operation in a clean and dry indoor laboratory environment by qualified personnel. The product or its external components shall not be exposed to corrosive and/or flammable substances, liquids or extreme heat. Do not operate with high dust and humidity levels.

Table 1: Operation environment conditions

Parameter	Value
Temperature	+5 °C to +45 °C
Relative humidity	< 80 %, no condensation
Maximum altitude	2000 meters
Protection level	IP 20 (IEC 60529)

10.1.3 Electrical characteristics

Table 2: Power supply ratings

Parameter	Value
Voltage	100 - 240 V AC
Frequency	47-63 Hz
Power	max. 60 W

The maximum voltage and current ratings of the signal inputs and outputs are specified on the *Time Tagger X*'s housing and must not be exceeded.

10.1.4 Electrostatic-sensitive device

The *Time Tagger X* is a sensitive electronic device and must be handled with care. The input and output circuitry of the *Time Tagger X* may be damaged by an electrostatic discharge or their functionality may be impaired. Take appropriate precautions to minimize the risk of an electrostatic discharge into the connections or signal wiring during installation and operation of the *Time Tagger X*.



The *Time Tagger X* is sensitive to electrostatic discharge! Take appropriate protective measures when performing system installation or maintenance.

10.1.5 Equipment installation

The *Time Tagger X* does not require assembly and is supplied fully assembled. It can be installed as a table top instrument or in a standardized 19-inch rack. Any operating position shall provide adequate space for cable connections and air circulation.

A table top installation requires an even and horizontal surface with enough space for easy access of the device.

Installation in a standardized 19-inch rack requires 2 height units. The rack shall provide sufficient ventilation and must not obstruct the ventilation openings of the *Time Tagger X*'s rear panel.

Position the *Time Tagger X* in a manner that allows the user to disconnect the device from the mains at any time and without restrictions.

Before connecting to the mains, inspect the product and the power cord for visible damage. Connection to the mains shall be done only with the supplied detachable power cord. In case of doubt, do not operate the product and seek for assistance from a qualified electrician or a person responsible for electrical safety.



Safe operation can no longer be assumed:

- after rough handling during transport or installation,
- in case of visible damage to the product or its power cord,
- in case of loose internal parts being noticed,
- in case of ingress of any liquids inside the product's housing.

The *Time Tagger X* provides one functional ground terminal on the rear panel located next to the power cord receptacle. All connectors that feature ground terminals are electrically connected to the *Time Tagger X* housing and to the functional ground terminal. This includes shield terminals on the SMA connectors and the USB-C connector.

Take care of appropriate grounding practices and ensure that all connected accessory and equipment shares common ground with the *Time Tagger X*. Connection of additional accessories or equipment that are grounded to an independent grounding circuit has the potential risk of electrical shock. An inappropriately grounded *Time Tagger X* and/or connected equipment may result in product damage, malfunction, degraded performance or poor signal quality.

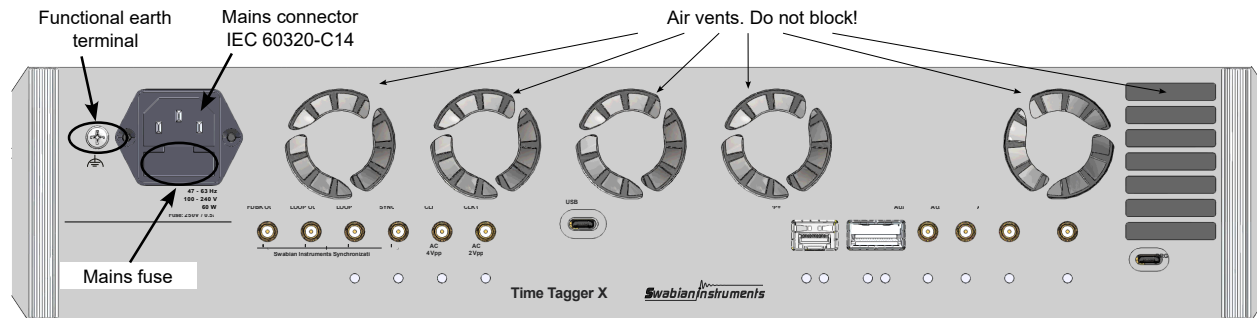


Fig. 1: View of the *Time Tagger X* rear panel.

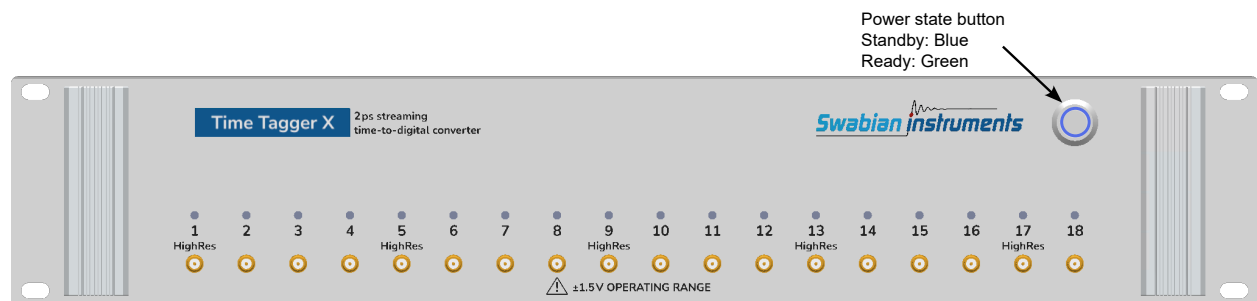


Fig. 2: View of the *Time Tagger X* front panel with power state button indicated.

After connecting the *Time Tagger X* to the mains, it will be in a “Standby” power mode indicated by the power button light pulsating blue. Press the power button to switch the product into the “Ready” state - the power button light will turn green. By pressing the power button again, the *Time Tagger X* can be switched back into the “Standby” power mode.

The *Time Tagger X* produces waste heat during regular operation. Long-term continuous operation at elevated environment temperatures may lead to a warm product surface, this is not a malfunction. Stop operation immediately and contact the manufacturer in case the product’s surface temperature exceeds 55 °C (hot on brief touch).

10.1.6 Maintenance and repair

The *Time Tagger X* does not contain any user serviceable or user repairable parts. In case of malfunction or damage, stop operating the *Time Tagger X*, disconnect it from the mains and contact the manufacturer.



The *Time Tagger X* may only be opened by personnel authorized by the manufacturer. Disconnect the *Time Tagger X* from the mains before opening it and working on the internal components. Otherwise the personnel may be exposed to the risk of electric shock. Adjustments, replacements of parts and repair may be carried out only by personnel authorized by the manufacturer.

All *Time Tagger X* units repaired by the manufacturer undergo verification and testing procedures in the same way as new units. The electrical safety of every new and repaired *Time Tagger X* is verified with a Gossen Metrawatt SECUTEST PRO testing instrument.

Power cord

The detachable power cord must be adequately rated for the operation voltage and power consumption of the *Time Tagger X*, see the table [Power supply ratings](#).



It is strictly forbidden to use a damaged power cord or a cord with inadequate ratings. In case of doubt, do not operate the product and contact the manufacturer or seek for assistance from a trained electrician or a person responsible for electrical safety.

Safety fuse

The *Time Tagger X* is equipped with a mains fuse located on the rear panel beneath the power cord receptacle. The fuse compartment can only be accessed when the power cord is detached from the *Time Tagger X*. The fuse type and fuse rating must comply with the specification in the table [Safety fuse ratings](#) and the labeling on the *Time Tagger X*.

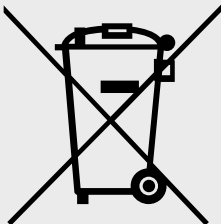
Table 3: Safety fuse ratings

Parameter	Value
Voltage Rating	250 V AC
Current Rating	500 mA
Dimensions	5 x 20 mm
Response Time	Slow Blow

Cleaning

Before cleaning the *Time Tagger X*, please make sure that it has been switched off and disconnected from the mains. Clean the outer housing with a soft, lint-free dust cloth. No parts of the *Time Tagger X* may be cleaned with chemical cleaning agents.

10.1.7 Disposal and recycling



Never dispose of the *Time Tagger X* with the household waste. Please inform yourself about the local rules on separate collection of electrical and electronic products.

10.1.8 Contact, support and service

Swabian Instruments GmbH can be reached directly via phone, email or per post.

Swabian Instruments GmbH
Stammheimer Straße 41
70435 Stuttgart
Germany

+49 711 400 479-0
info@swabianinstruments.com
<https://www.swabianinstruments.com>

REVISION HISTORY

11.1 V2.17.4 - 17.07.2024

Improvement

- Python: adds support for numpy 2.0.0.

11.2 V2.17.2 - 02.07.2024

Improvement

- Adds support for Ubuntu Long Term Support release 24.04.
- Adds *IteratorBase.abort()* for a non-blocking hint to abort measurements quickly.
- Adds a tutorial for ODMR measurements.

Time Tagger bug fixes

- Fixes wrong initial UTC time stamp in *PulsePerSecondMonitor* and improved formatting in its data export file.
- Fixes *HistogramND* for $N > 4$.
- Fixes a bug with missing time tags when multiple small saved files are merged and replayed.

Time Tagger Lab bug fixes

- Fixes a crash when switching between *FileWriter* and other measurements.
- Adds missing data export option for *PulsePerSecondMonitor* and *FrequencyCounter*.
- Fixes visualization of available HighRes channels for *Time Tagger X* on the landing page.
- Fixes a bug when stopping and restarting the Logarithmic Histogram DLS simulation.
- Fixes a crash in the application's window docking manager.
- Includes several smaller UI and stability improvements.

11.3 V2.17.0 - 22.04.2024

Highlights

- With the new *FrequencyCounter* measurement, the Time Tagger becomes a full feature Omega-type frequency counter.
- The new *PulsePerSecondMonitor* measurement allows to monitor the synchronicity of different PPS sources.
- The support of variable integration time per bin in *HistogramLogBins* provides the accurate g2 normalization with immediate start and gated inputs.
- Improve the performance of the Synchronizer to reach the full 80 MTags/s per device.

Improvement

- New virtual channel *Combinations* for exclusive coincidences.
- Adds support for the Python Stable ABI to support Python 3.12 and likely many more further releases.
- Adds support for the MinGW C++ ABI for the *MINGW32* and *UCRT64* environment.

Time Tagger Ultra and Time Tagger X

- Significantly reduced on-device latency noticeable both over USB and FPGA link.
- Support for *high priority input channels*, which will still be transmitted while in overflow domain.
- The timestamps of rising and falling events can be *averaged on hardware* for a higher precision of events.
- Enhanced version of the *deadtime filter*, which can be configured with any dead time in picosecond precision.

Time Tagger X

- Full support for the *High-Resolution* mode on the *Time Tagger X* with a timing jitter of 1.5 ps RMS per channel.
- Support for the QSFP+ *FPGA link* with a data rate of 1200 MTags/s.
- Support for the readout of more device sensors on the hardware.
- The self verification of the hardware input stage of the built-in test signal is reverted on the *Time Tagger X*.
- All input ports stay at the *high-impedance* mode before their first usage and after *freeTimeTagger()*.
- Changes the default *hysteresis* from 1 mV to 20 mV.

Time Tagger Ultra

- Adds support for *Time Tagger Ultra* with hardware revision 1.8.
- The improved auto-calibration for periodic signals and its error handling is backported from the *Time Tagger X* to the *Time Tagger Ultra*.
- Improves the USB performance for many aligned inputs.

Time Tagger Lab features and enhancements

- Improved and more informative landing page layout.
- Existing measurements can now be cloned easily (by right clicking).
- The live countrate is now visible in the detailed device view.
- Application window positions and the measurement list order are now saved to workspace.
- A notification appears when the measurement dead time is automatically adjusted to a multiple of the Time Tagger clock.
- New startup command line options `--select-device` and `--start-measurement`.
- Maximum file size for the *FileWriter* measurement is now shown in megabytes.
- “Force high impedance” option for *Time Tagger X* start-up without impedance switch.
- Many smaller UI improvements and fixes.

Time Tagger Lab bug fixes

- Fixed various crashes related to device license checks, sending feedback and exporting large data sets to file.
- Fixed occasional misaligned axis ticks and missing axis ticks when zooming in.
- Fixed limited application visibility for high screen scaling settings.
- Fixed occasionally missing chart cursor values.
- Changed logarithmic axis labels from $E^$ to $10^$.

11.4 V2.16.2 - 28.06.2023

Time Tagger hardware support

- Adds support for *Time Tagger 20* with hardware revision 2.5.
- Adds support for *Time Tagger X* with hardware revision 1.2.

Time Tagger bug fix

- Fixes *Counter* showing wrong values when the same channel is used multiple times, now throwing an exception on creating.
- Fixes *FileReader* in Matlab not loading the .NET Assembly in its constructor. All measurement classes now load the assembly.

Time Tagger Lab

- Fixes non-visible numbers for cursor values.
- Moved legend of Counter time trace to the top left.
- Higher precision formatting values when necessary.

11.5 V2.16.0 - 05.06.2023

Highlights

- Time Tagger Lab: Virtual Antibunching and Fluorescence lifetime setups.

Time Tagger Lab features and enhancements

- Antibunching and Fluorescence lifetime setups and corresponding simulation measurements. The detector signals are fully simulated for those measurements and no physical detectors are needed.
- Option to suppress counts for $dt=0$ with autocorrelation measurements.
- Notification that software updates are available.
- Improved crash and error reporting.
- Improved ticks and labels of logarithmic plots.

Time Tagger Lab bug fixes

- Fixed various crashes related to exporting data, opening a message box, and creating log files.
- Fixed Histogram 2D export.
- Fixed auto-restarting with no-plot measurements on configuration change.
- Missing log-y labels and ticks are added to chart display.

Improvement

- Adds support for Python 3.11.
- The built-in test signal verifies the hardware input stage on *Time Tagger X* starting from hardware revision 1.1.

Bug fixes

- Fixes fan information of `getSensorData()` on the *Time Tagger Ultra*.
- Fixes the support for *Time Tagger Ultra* devices with the serials starting with 17.
- Fixes `GatedChannel`, `FrequencyMultiplier` and `TriggerOnCounterRate` to switch to the initial state on `clear()` and `startFor()`.
- Fixes `autoCalibration()` to clear the old calibration data.
- Fixes the error handling on Linux if parts of the Python wrapper are missing.
- Fixes a one bit out of bounds memory access on verifying the hardware license.

- Fixes the jagged array handling of the Matlab Wrapper for *Coincidences*.

11.6 V2.15.0 - 06.03.2023

Highlights

- Rework of the Matlab wrapper, now reflecting the full functionality of the Time Tagger API.

Matlab wrapper

- Supports class inheritance, new TT classes are closer to their C++, and Python counterparts.
- Supports creation of TimeTagger objects via *TimeTagger.createTimeTagger()*.
- Supports native Matlab enums.
- Supports jagged arrays via conversion from/into Matlab cells.

Behavior change

- Disallow an event divider value of 0. Please use 1 instead.
- Matlab: Removed Matlab app designer examples.

Other Changes

- Updated Opal Kelly Frontpanel library to 5.2.12.

Bug fixes

- Python: Fix return type annotation for *createTimeTagger()*.
- Fixes certain deadlock issues in Matlab caused by asynchronous functions.
- Fixes the overflow tracking with a USB error while in overflow.
- Fixes handling of tags with respect to <-> missed events at the timestamp of an OverflowBegin error.
- Counter: Fixes a transpose issue of the NaN generation in *CounterData.getDataNormalized()*.
- FrequencyStability: Fixes an integer overflow in the MDEV / TDEV calculation.
- Fixed issue with *LicenseRequestGenerator.exe* not working on some systems.

Time Tagger Lab features and enhancements

- Device LEDs can be switched on and off (not *Time Tagger 20*).
- Improved setting of the reference software clock for signals with an event divider greater than 1.
- *Time Tagger 20* Edu is supported.

Time Tagger Lab bug fixes

- Fix the crash when capturing high-resolution counter-time traces.
- Fix the crash when the USB connection to the device is unstable.
- Fix the crash when exporting large Scope measurement data.
- Fix the crash when exporting a plot to a PNG file.
- Fix the crash when measurements require too much memory on initialization.
- User interface issues with small screens are solved.

11.7 V2.14.0 - 23.12.2022

Time Tagger X

- Adds support for *Time Tagger X* with hardware revision 1.1.
- Adds support for Synchronizer with *Time Tagger X* and also mixed setups with *Time Tagger Ultra*.
- Provides preliminary Ethernet support on the SFP+.
- Improves the auto-calibration for periodic signals.
- Better CPU performance for aligned tags.
- Improves the channel LED blinking for 1pps signals.
- Shows calibration errors on the channel LED in red.
- Fixes a random skew jump of 1333 ps per input channel on USB errors.
- Fixes the channel LED indication for falling events.

Bug fix

- Fix Power LED turning off after calling `freeTimeTagger()` on the *Time Tagger Ultra*.

11.8 V2.13.2 - 22.11.2022

Highlight

- Release of *Time Tagger Lab* - the native UI for Windows.

11.9 V2.12.4 - 09.11.2022

Improvement

- Adds support for *Time Tagger Ultra* with hardware revision 1.7.

Bug fix

- Fixed *Web Application* data export for *HistogramLogBins*.

11.10 V2.12.2 - 04.10.2022

Bug fixes

- Fixes wrong *Error* event generation and channel detection in *mergeStreamFiles()*.
- Fixes host license querying for *TimeTaggerNetwork* clients (broken in v2.12.0)
- Python: Fixes the decoding of the return values of *getDeviceLicense()* and *getSensorData()* of *TimeTaggerNetwork*.
- C++: Fixes `std::string` unmarshalling within the Network Time Tagger.

Various improvements

- Adds method *getTraceFrequencyAbsolute()* to *FrequencyStability*.
- Adds method *getChannellist()* to *FileReader*.

11.11 V2.12.0 - 01.09.2022

Highlights

- Add support for our new high precision measurement device, the *Time Tagger X*, with a typical timing resolution of 2.0 picoseconds.

WebApp

- Change default access mode to *AccessMode.Control* for Time Tagger Network.

Features

- Improve the performance of the Synchronizer. Two *Time Tagger Ultra* devices now can achieve a total data rate of over 100 Mtags per second.
- Add support for the new Ubuntu Long Term Support release 22.04 Jammy Jellyfish.

Behavior change

- All provided strings to the C++ API must be encoded as UTF-8, returning strings are also UTF-8 encoded.
- Rename `getLicenseInfo()` to *TimeTagger.getDeviceLicense()*. It returns a JSON formatted string for easier processing.
- Drop support for obsolete Python versions 2.7 and 3.5 and for the obsolete Linux distributions CentOS 7 and Ubuntu 16.04.

Examples

- Jitter verification requests the specified jitter values directly from the Time Tagger backend.
- FLIM example is now available for Matlab

Various Fixes and Improvements

- Fix Unicode characters in all filenames of *FileWriter* and *FileReader*.
- Fix CoincidenceFactory for Matlab and Labview.
- Fix the *Flim* measurement for Matlab.
- Fix the crash on a failing license download within the initialization of the Network Time Tagger.
- Prefer a host-locked license over a user-locked license in the Virtual Time Tagger. This reduces the chance of a false-positive anti-virus warning.
- Improve the rounding behavior of *TimeTagger.getTriggerLevel()*.
- Update of the USB driver for various fixes.

11.12 V2.11.0 - 22.04.2022

Highlights

- Introduced *mergeStreamFiles()* to combine several *FileWriter* files into one.

Time Tagger Network

- New Protocol version 3.1 with a new set of features. Backward compatible with 3.0.
- Improved messages for connection loss and disconnection. Additionally, messages for connecting to new and old versions of TTN will be presented.
- Fixed issue related to implicit call of `sync()` on measurement creation in `AccessMode.Listen` mode.
- Server and network information has been added to `getConfiguration()`.
- Fixed crashes when streaming over 250 channels.
- Various race conditions and possible freezes have been fixed.
- Faster initialization of measurements with many channels.
- Fixed error handling on disconnection.
- Reduced connection timeout to 10 seconds.
- Fixed an issue where channels used by a client remained registered after a disconnect.
- `TimeTagger.setTimeTaggerNetworkStreamCompression()` can be utilized to double the maximum transfer rate in a very slow network environment (≤ 100 Mbit/s).

Time Tagger Virtual

- The `TimeTaggerVirtual` will now wait for a test signal channel to be registered before starting to stream it (behavior identical to a hardware Time Tagger).

GatedChannel

- Optional constructor argument `initial` of type `GatedChannelInitial` to initialize the gate optionally in an open state.
- Changed behavior if `input_channel` equals `gate_start_channel` or `gate_stop_channel` to allow for operation similar to the ConditionalFilter.

FrequencyStability

- Fixed `getTraceFrequency()`; now it returns the relative frequency error instead of the relative period error.
- Traces are no longer truncated to the length of the maximum steps.
- Corrected behavior if stopped and restarted without clearing.

Other measurement classes

- `HistogramLogBins`: Removed bins which have a bin width of 0 ps.
- `SynchronizedMeasurements`: Methods calls on a `SynchronizedMeasurements` object without any registered measurements will no longer generate an exception but a warning.
- `TimeDifferences`: Added `getHistogramIndex()` to return the index of the histogram being processed currently.
- Exposed `TagType` to be used with `TimeTagStream` and `CustomMeasurement`.

Synchronizer

- Improved error messages.
- Fixed USB errors occurring under very high data rates.

Examples

- Added Visual Basic .NET example

Various Fixes

- Fixed crash on `createTimeTagger()` during a USB error.
- Fixed an issue where `startFor()` could run further than the specified time on `HistogramLogBins` and `FileWriter`.
- WebApp: Fixed argument handling on Linux.
- Matlab: Supports now `Resolution` for HighRes.
- Matlab: Verifies that the version of the installed backend matches the wrapper version.

11.13 V2.10.6 - 16.03.2022

Improvements

- Adds support for *Time Tagger Ultra* with hardware revision 1.6b.

11.14 V2.10.4 - 23.02.2022

WebApp

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

11.15 V2.10.2 - 31.12.2021

Improvements

- Added support for Python 3.10.

Fixes for issues since v2.10.0

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

11.16 V2.10.0 - 22.12.2021

Highlights

- Time Tagger Network: All Time Tagger devices and the acquired data can be accessed via the network from multiple clients or locally across the different programming languages. The clients can use all TimeTagger measurement classes and may optionally control the settings of the physical Time Tagger.
- A new frequency stability toolbox: It offers on-the-fly evaluation of periodic signals by calculating several analysis metrics, including, for example, the Allan deviation (ADEV) and time deviation (TDEV).
- Software Clock: The new recommended method for using an external clock on the *Time Tagger Ultra*. The time tag stream is rescaled on the software side with respect to the connected clock. It allows for a broad input frequency range and also calculates phase error estimators. In addition, the input jitter of the clock channel will be averaged out, resulting in a lower jitter for measurements including the clock channel directly.

Features

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

WebApp

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

Performance

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

Behavior change

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

Examples

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

Fixes

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

11.17 V2.9.0 - 07.06.2021

Highlights

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

Changes

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

Features

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

Bug fixes

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

11.18 V2.8.4 - 04.05.2021

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

11.19 V2.8.2 - 26.04.2021

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

11.20 V2.8.0 - 29.03.2021

Highlights

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

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

New Time Tagger Ultra features

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

Changes

- Split `TimeTaggerBase.setInputDelay()` into `TimeTaggerBase.setDelayHardware()` and `TimeTaggerBase.setDelaySoftware()`.
- `TimeTagger.getChannelList()` filter enum renamed to `ChannelEdge`.
- `TimeTagger.setNormalization()` can now be configured per channel.
- Changed the default port of the WebApp to 50120 to avoid collision with Jupiter Notebooks.
- Maximum input frequency of the *Time Tagger Ultra* is reduced to 475 MHz.
- The deadtime specification of the *Time Tagger Ultra* changed to 2.1 ns. It can detect events separated by 2 ns with possible loss of some events.

Features

- Added a *TriggerOnCountRate* virtual channel that generates events when a count rate crosses the given threshold value.
- Added support for Python 3.9.
- *waitUntilFinished()* and *sync* have an optional timeout parameter.

Examples

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

Bug fixes

- Histogram can be used with *waitUntilFinished()* and *SynchronizedMeasurements*. *Histogram* is now derived from *IteratorBase()*.
- Displaying the singleton warning of *createTimeTagger* just once.
- Fixed string conversion issue for old Matlab versions.
- Hide “unused argument” warnings in the TimeTagger C++ headers.

11.21 V2.7.6 - 26.04.2021

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

11.22 V2.7.4 - 19.04.2021

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

11.23 V2.7.2 - 22.12.2020

Highlights

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

Improvements

- Added AnyCPU targeted .NET Assembly for C# wrappers. Available in GAC_MSIL and the installation folder.
- More detailed error handling and human-readable error messages.
- Added *Conditional Filter* for *TimeTaggerVirtual*.
- Removed Intel's *libmmd.dll* library dependency.
- All measurements have the new common method *waitUntilFinished()*, which can be used with *startFor()*.
- Warnings are printed with time information.
- Cleanup of the C++ measurements' header file.
- Remote license upgrades can be performed via the web application.
- Reworked Python and C# examples.

Fixes

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

API changes

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

11.24 V2.7.0 - 01.10.2020

Highlights

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

Fixes and improvements

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

11.25 V2.6.10 - 07.09.2020

Fixes and improvements

- Fixes input delay, deadtime and test signal generator for the TimeTaggerVirtual.
- Fixes `getInvertedChannel` with the Swabian Synchronizer and with *Time Tagger Ultra* 8 devices with the old channel numbering schema.
- x axis is zoomable with Scope measurement.
- Better error handling for non-existent files with TimeTaggerVirtual and FileReader.

Python

- Changed the constants `CoincidenceTimestamp_` to a Python enum (e.g., `CoincidenceTimestamp_First` is now `CoincidenceTimestamp.First`).

Matlab

- Enum for timestamp argument for `Coincidence(s)` is available via `TTCoincidenceTimestamp`.

Linux

- Fix for slow Linux device opening.

11.26 V2.6.8 - 21.08.2020

Highlights

- Support for the Time Tagger Value edition. This is an upgradeable and cost-efficient version of the *Time Tagger Ultra* for applications with moderate timing precision requirements.

Webapp

- Added *Histogram2D* to the measurement list.
- Improved performance and responsiveness for large datasets.
- 32-bit version of the Web Application works again.
- Fixed a bug that data of stopped measurements could not be saved.
- Fixed a bug that settings saved had the file extension .json instead of .ttconf ending.
- Fixed a bug when using falling edges for Time Tagger starting with channel 0.

Python

- Fixed a bug that some named arguments could not be used anymore.

API

- Added the method *SynchronizedMeasurements.unregisterMeasurement()* to remove measurements from *SynchronizedMeasurements*.

Backend

- Improved performance of the FileWriter, exceeding 100 M tags/s on high-end CPUs.
- Improved binning performance of all histogram measurements: Correlation, FLIM, Histogram, StartStop, TimeDifferences, TimeDifferencesND.
- Fixes a deadlock in the virtual Time Tagger if a measurement accesses some public methods of the Time Tagger.

11.27 V2.6.6 - 10.07.2020

Highlights

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

Webapp

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

Time Tagger Ultra

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

Backend

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

11.28 V2.6.4 - 27.05.2020

WebApp

- Option to enable logarithmic y-axis scaling for Counter, Histogram, HistogramLogBins and Correlation
- Redesign "Create measurement" dialog with links to the online documentation
- Fixed flickering when switching between plots
- Fixed plotting wrong data range when changing the number of data points
- Added the basic functionality of the TimeTaggerVirtual (test signal only)

New features and improvements

- Added the test signal to TimeTaggerVirtual
- Support for Ubuntu 20.04 and CentOS 8
- LabVIEW example for FileWriter and FileReader
- Improved Matlab API for VirtualTimeTagger, adding optional parameters
- Make the data transfer size configurable by .setStreamBlockSize
- Performance improvements for HistogramLogBins
- Slightly improved timing jitter at large time differences for the *Time Tagger 20*
- Time Tagger Application works again with 32 bit operating systems
- Connection errors are shown in the Matlab console or can be handled with the new logger functionality
- Added custom logger examples for Matlab/Python/C#

Changes

- Updated the USB library
- Stop measurements when freeTimeTagger is called (e.g. closes files on dump, isRunning now returns false)
- Reduced polling rate (0.1s) for USB reconnections

API changes

- Added .setLogger() to attach a callback function for custom info/error logging
- Rename of enumeration ErrorLevel to LogLevel
- Rename of log level constants and with new corresponding integer values

11.29 V2.6.2 - 10.03.2020

Highlights

- TimeTaggerVirtual, FileWriter, and FileReader have reached a stable state
- Improved Linux support (documentation, compiling custom Python wrappers)

New features

- Added setInputDelay, setDeadtime, getOverflows, and more to the TimeTaggerVirtual
- Add an optional parameter in setConditionalFilter for disabling the hardware delay compensation
- Infinite dumping in Dump for negative max_count
- Create a freeAllTimeTagger() method, which is called by Python atexit
- Reimplement SynchronizedMeasurements as a proxy tagger object, which auto registers new measurements without starting them

- The new `SynchronizedMeasurements.isRunning()` method returns if any measurement is still running
- Python: Distribute the generated C++ wrapper source for supporting future Python revisions
- C++: New `IteratorBase.getLock` method returning a `std::unique_lock`
- C++: Improved exception handling for custom measurements: exceptions now stop the measurement, `runSynchronized` forwards exceptions to the caller

API changes

- `TimeTagger.getVersion` return value is changed to a string
- C++: Use 64 bit integers for the dimensions in the `array_out` helpers
- C++: Rename the base class for custom measurements from `_Iterator` to `IteratorBase`
- C++: Constructors of custom measurements shall call `finishInitialization` instead of `IteratorBase.start`
- Python 2.7: Update the numpy C headers to 1.16.1

Examples and documentation

- Improved Histogram2D example
- Clarify `setInputDelay` vs `DelayedChannel`

Bug fixes

- Relax the voltage supply check in the *Time Tagger Ultra* hardware revision 1.4
- Use a 1 MB buffer for `Dump`, `FileWriter`, and `FileReader` to achieve full speed especially on network devices
- Fix `getTimeTaggerModel` on an active device
- Fix deadlock within `sync()` while the device is disconnected
- Provide the documentation on Linux
- Several fixes and improvements for the `FileWriter` and `TimeTaggerVirtual`

WebApp

- Improved default names for measurements
- Not relying on data stored within the browser any more
- Disabling mouse scrolling within numeric inputs
- Various buxfixes

11.30 V2.6.0 - 23.12.2019

Highlights

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

Known issues

- FileWriter and FileReader have a low performance on network devices

API changes

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

New features

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

- support for the *Time Tagger Ultra* hardware revision 1.4

Examples

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

Bug fixes

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

11.31 V2.4.4 - 29.07.2019

- reduced crosstalk between nonadjacent channels of the *Time Tagger Ultra*
- fixed a bug leading to high crosstalk with V2.4.2 for specific channels
- fixed a rare clock selection issue on the *Time Tagger 20*
- improved and more detailed documentation
- new method `Countrate.getCountsTotal()`, which returns the absolute number of events counted
- new Mathematica quickstart example
- new *Scope* example for LabVIEW
- support of the *Time Tagger 20* series with hardware revision 2.3

- release the Python GIL while in the Time Tagger engine code
- fixed a bug in *ConstantFractionDiscriminator*, which could cause that no virtual tags were generated

11.32 V2.4.2 - 12.05.2019

- support of the *Time Tagger Ultra* series with hardware revision 1.3
- improve performance of short pulse sequences on the *Time Tagger 20* series
- improve overflow behavior at too high input data rates
- fix the name of the ‘SynchronizedMeasurements’ measurement class

11.33 V2.4.0 - 10.04.2019

Libraries

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

API

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

Examples

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

Time Tagger Ultra

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

Time Tagger 20

- 166.6 MHz EXT input clock detection enabled

Operating systems

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

11.34 V2.2.4 - 29.01.2019

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

11.35 V2.2.2 - 13.11.2018

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

11.36 V2.2.0 - 07.11.2018

General improvements

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

API

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

Web application

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

11.37 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

11.38 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

11.39 V2.1.2 - 14.03.2018

fixed issue installing the Matlab toolbox

11.40 V2.1.0 - 06.03.2018

Time Tagger Ultra

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

11.41 V2.0.4 - 01.02.2018

Bug fixes

- Closing the web application server window works properly now

11.42 V2.0.2 - 17.01.2018

Improvements

- Matlab GUI example added
- Matlab dump/load example added

Bug fixes

- dump class writing tags multiple times when the optional channel parameter is used
- Counter and Countrate skip the time in between a .stop() and a .start() call
- The Counter class now handles overflows properly. As soon as an overflow occurs the lost data junk is skipped and the Counter resumes with the new tags arriving with no gap on the time axis.

11.43 V2.0.0 - 14.12.2017

Release of the Time Tagger Ultra

Note: The input delays might be shifted (up to a few hundred ps) compared to older driver versions.

Documentation changes

- new section 'In Depth Guides' explaining the hardware event filter

Webapp

- fixed a bug setting the input values to 0 when typing in a new value
- new server launcher screen which stops the server reliably when the application is closed

11.44 V1.0.20 - 24.10.2017

Virtual Channels

- DelayedChannel clones and optionally delays a stream of time tags from an input channel
- GatedChannel clones an input stream, which is gated via a start and stop channel (e.g. rising and falling edge of another physical channel)

API

- startFor(duration) method implemented for all measurements to acquire data for a predefined duration
- getCaptureDuration() available for all measurements to return the current capture duration
- getDataNormalized() available for Correlation
- setEventDivider(channel, divider) also transmits every nth event (divider) on channel defined

Webapp

- label for 0 on the x-axis is now 0 instead of a tiny value

C++ API:

- internal change so that clear_impl() and next_impl() must be overwritten instead of clear() and next()

Other bug fixes/improvements

- improved documentation and examples

11.45 V1.0.6 - 16.03.2017

Web application (GUI)

- load/save settings available for the Time Tagger and the measurements
- correct x-axis scaling
- input channels can be labeled
- save data as tab separated output file (for Matlab, Excel, ... import)
- fixed: saving measurement data now works reliably
- fixed: 'Initialize' button of measurements works now with tablets and phones

API

- direct time stream access possible with new class TimeTagStream (before the stream could be only dumped with Dump)
- Python 3.6 support
- better error handling (throwing exceptions) when libraries not found or no Time Tagger attached
- setTestSignal(...) can be used with a vector of channels instead of a single channel only
- Dump(...) now with an optional vector of channels to explicitly dump the channels passed
- CHANNEL_INVALID is deprecated - use CHANNEL_UNUSED instead
- Coincidences class (multiple Coincidences) can be used now within Matlab/LabVIEW

Documentation changes

- documentation of every measurement now includes a figure
- update and include web application in the quickstart section

Other bug fixes/improvements

- no internal test tags leaking through from the initialization of the Time Tagger
- Counter class not clearing the data buffer in time when no tags arrive
- search path for bitfile and libraries in Linux now work as they should
- installer for 32 bit OS available

11.46 V1.0.4 - 24.11.2016

Hardware changes

- extended event filter to multiple conditions and filter channels
- improved jitter for channel 0
- channel delays might be different from the previous version (< 1 ns)

API changes

- new function setConditionalFilter allows for multiple filter and event channels (replaces setFilter)
- Scope class implements functionality to use the Time Tagger as a 50 GHz digitizer
- Coincidences class now can handle multiple coincidence groups which is much faster than multiple instances of Coincidence
- added examples for C++ and .net

Software changes

- improved GUI (Web application)

Bug fixes

- Matlab/LabVIEW is not required to have the Visual Studio Redistributable package installed

11.47 V1.0.2 - 28.07.2016

Major changes:

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

API changes:

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

11.48 V1.0.0

initial release supporting python

11.49 Channel Number Schema 0 and 1

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

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

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

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

With release V2.2.0, the channel number was detected automatically for the device in use. It was according to the labels on the device. With release V2.17.0, the channel number starts with the number 1 by default for all devices, regardless of the labels on the device.

In case another channel schema is required, please use `setTimeTaggerChannelNumberScheme(int scheme)` before the first Time Tagger is initialized.

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

A

abort() (*IteratorBase* method), 108
 AccessMode (*built-in class*), 68
 align_to_reference (*FrequencyCounterData* attribute), 142
 All (*ChannelEdge* attribute), 69
 autoCalibration() (*TimeTagger* method), 85
 Average (*CoincidenceTimestamp* attribute), 69
 averaging_periods (*SoftwareClockState* attribute), 95

B

bins (*FlimFrameInfo* attribute), 133
 built-in function
 createTimeTagger(), 71
 createTimeTaggerNetwork(), 71
 createTimeTaggerVirtual(), 71
 freeTimeTagger(), 72
 getTimeTaggerChannelNumberScheme(), 73
 getTimeTaggerServerInfo(), 72
 getUsageStatisticsReport(), 75
 getUsageStatisticsStatus(), 74
 getVersion(), 74
 mergeStreamFiles(), 73
 scanTimeTagger(), 72
 scanTimeTaggerServers(), 72
 setLogger(), 73
 setTimeTaggerChannelNumberScheme(), 73
 setUsageStatisticsStatus(), 74

C

CHANNEL_UNUSED (*built-in variable*), 68
 ChannelEdge (*built-in class*), 69
 ChannelGate (*built-in class*), 75
 clear() (*Dump* method), 151
 clear() (*IteratorBase* method), 108
 clear() (*SynchronizedMeasurements* method), 156
 clearConditionalFilter() (*TimeTagger* method), 82
 clearOverflows() (*TimeTaggerBase* method), 78
 clearOverflowsClient() (*TimeTaggerNetwork* method), 94
 clock_period (*SoftwareClockState* attribute), 95
 Closed (*GatedChannelInitial* attribute), 70

Coincidence (*built-in class*), 97
 Coincidences (*built-in class*), 99
 CoincidenceTimestamp (*built-in class*), 69
 Collecting (*UsageStatisticsStatus* attribute), 71
 CollectingAndUploading (*UsageStatisticsStatus* attribute), 71
 Combinations (*built-in class*), 100
 Combiner (*built-in class*), 101
 ConstantFractionDiscriminator (*built-in class*), 102
 Control (*AccessMode* attribute), 68
 Correlation (*built-in class*), 123
 CountBetweenMarkers (*built-in class*), 114
 Counter (*built-in class*), 111
 CounterData (*built-in class*), 112
 Countrate (*built-in class*), 110
 createTimeTagger()
 built-in function, 71
 createTimeTaggerNetwork()
 built-in function, 71
 createTimeTaggerVirtual()
 built-in function, 71
 CustomMeasurement (*built-in class*), 157

D

DelayedChannel (*built-in class*), 102
 Disabled (*UsageStatisticsStatus* attribute), 71
 disableFpgaLink() (*TimeTagger* method), 88
 disableLEDs() (*TimeTagger* method), 87
 disableSoftwareClock() (*TimeTaggerBase* method), 79
 dropped_bins (*CounterData* attribute), 112
 Dump (*built-in class*), 151

E

enabled (*SoftwareClockState* attribute), 95
 enableFpgaLink() (*TimeTagger* method), 88
 Error (*TagType* attribute), 70
 error_counter (*SoftwareClockState* attribute), 95
 Event (*built-in class*), 153
 EventGenerator (*built-in class*), 103
 External delay, 162

F

Falling (*ChannelEdge* attribute), 69
 FileReader (*built-in class*), 150
 FileWriter (*built-in class*), 149
 First (*CoincidenceTimestamp* attribute), 69
 Flim (*built-in class*), 128
 FlimBase (*built-in class*), 134
 FlimFrameInfo (*built-in class*), 132
 FpgaLinkInterface (*built-in class*), 69
 frame_number (*FlimFrameInfo* attribute), 133
 frameReady() (*Flim* method), 132
 frameReady() (*FlimBase* method), 135
 freeTimeTagger()
 built-in function, 72
 FrequencyCounter (*built-in class*), 141
 FrequencyCounterData (*built-in class*), 142
 FrequencyMultiplier (*built-in class*), 104
 FrequencyStability (*built-in class*), 137
 FrequencyStabilityData (*built-in class*), 137

G

GatedChannel (*built-in class*), 104
 GatedChannelInitial (*built-in class*), 70
 getADEV() (*FrequencyStabilityData* method), 138
 getADEVScaled() (*FrequencyStabilityData* method), 139
 getBinEdges() (*HistogramLogBins* method), 119
 getBinWidths() (*CountBetweenMarkers* method), 115
 getCaptureDuration() (*IteratorBase* method), 109
 getChannel() (*Combinations* method), 100
 getChannel() (*VirtualChannel* method), 97
 getChannelAbove() (*TriggerOnCountrate* method), 106
 getChannelBelow() (*TriggerOnCountrate* method), 106
 getChannelList() (*FileReader* method), 151
 getChannelList() (*TimeTagger* method), 85
 getChannels() (*TimeTagStreamBuffer* method), 148
 getChannels() (*TriggerOnCountrate* method), 106
 getChannels() (*VirtualChannel* method), 97
 getCombination() (*Combinations* method), 100
 getConditionalFilterFiltered() (*TimeTagger* method), 82
 getConditionalFilterTrigger() (*TimeTagger* method), 82
 getConfiguration() (*FileReader* method), 151
 getConfiguration() (*IteratorBase* method), 109
 getConfiguration() (*TimeTaggerBase* method), 80
 getConfiguration() (*TimeTaggerVirtual* method), 93
 getCounts() (*HistogramLogBinsData* method), 120
 getCounts() (*TimeDifferences* method), 126
 getCountsTotal() (*Countrate* method), 110
 getCurrentCountrate() (*TriggerOnCountrate* method), 106
 getCurrentFrame() (*Flim* method), 129
 getCurrentFrameEx() (*Flim* method), 130
 getCurrentFrameIntensity() (*Flim* method), 130
 getDACRange() (*TimeTagger* method), 85
 getData() (*Correlation* method), 123
 getData() (*CountBetweenMarkers* method), 114
 getData() (*Counter* method), 111
 getData() (*CounterData* method), 113
 getData() (*Countrate* method), 110
 getData() (*FileReader* method), 150
 getData() (*Histogram* method), 117
 getData() (*Histogram2D* method), 121
 getData() (*HistogramLogBins* method), 119
 getData() (*HistogramND* method), 122
 getData() (*Sampler* method), 154
 getData() (*Scope* method), 152
 getData() (*StartStop* method), 116
 getData() (*TimeDifferences* method), 125
 getData() (*TimeTagStream* method), 147
 getDataAsMask() (*Sampler* method), 155
 getDataNormalized() (*Correlation* method), 123
 getDataNormalized() (*Counter* method), 112
 getDataNormalized() (*CounterData* method), 113
 getDataNormalizedCountsPerPs() (*HistogramLogBins* method), 119
 getDataNormalizedG2() (*HistogramLogBins* method), 119
 getDataObject() (*Counter* method), 112
 getDataObject() (*FrequencyCounter* method), 142
 getDataObject() (*FrequencyStability* method), 137
 getDataObject() (*HistogramLogBins* method), 119
 getDataObject() (*PulsePerSecondMonitor* method), 145
 getDataTotalCounts() (*Counter* method), 112
 getDataTotalCounts() (*CounterData* method), 113
 getDeadtime() (*TimeTaggerBase* method), 77
 getDelayClient() (*TimeTaggerNetwork* method), 94
 getDelayHardware() (*TimeTaggerBase* method), 76
 getDelaySoftware() (*TimeTaggerBase* method), 77
 getDeviceLicense() (*TimeTagger* method), 87
 getDistributionCount() (*TimeTagger* method), 85
 getDistributionPSec() (*TimeTagger* method), 86
 getEventDivider() (*TimeTagger* method), 82
 getEventTypes() (*TimeTagStreamBuffer* method), 148
 getFence() (*TimeTaggerBase* method), 79
 getFrameNumber() (*FlimFrameInfo* method), 133
 getFramesAcquired() (*Flim* method), 130
 getFrequency() (*CounterData* method), 113
 getFrequency() (*FrequencyCounterData* method), 143
 getFrequencyInstantaneous() (*FrequencyCounterData* method), 143
 getG2() (*HistogramLogBinsData* method), 120
 getG2Normalization() (*HistogramLogBinsData* method), 120

getHardwareDelayCompensation() (*TimeTagger* method), 81
 getHDEV() (*FrequencyStabilityData* method), 138
 getHDEVsScaled() (*FrequencyStabilityData* method), 140
 getHistogramIndex() (*TimeDifferences* method), 125
 getHistograms() (*FlimFrameInfo* method), 133
 getIndex() (*Correlation* method), 124
 getIndex() (*CountBetweenMarkers* method), 114
 getIndex() (*Counter* method), 112
 getIndex() (*CounterData* method), 112
 getIndex() (*Flim* method), 130
 getIndex() (*FrequencyCounterData* method), 142
 getIndex() (*Histogram* method), 117
 getIndex() (*Histogram2D* method), 121
 getIndex() (*HistogramND* method), 122
 getIndex() (*TimeDifferences* method), 125
 getIndex_1() (*Histogram2D* method), 121
 getIndex_2() (*Histogram2D* method), 121
 getIndices() (*PulsePerSecondData* method), 145
 getInputDelay() (*TimeTaggerBase* method), 76
 getInputHysteresis() (*TimeTagger* method), 83
 getInputImpedanceHigh() (*TimeTagger* method), 83
 getIntensities() (*FlimFrameInfo* method), 133
 getInvertedChannel() (*TimeTaggerBase* method), 80
 getMaxFileSize() (*FileWriter* method), 149
 getMDEV() (*FrequencyStabilityData* method), 138
 getMissedEvents() (*TimeTagStreamBuffer* method), 148
 getModel() (*TimeTagger* method), 84
 getNormalization() (*TimeTagger* method), 84
 getOverflowMask() (*CounterData* method), 113
 getOverflowMask() (*FrequencyCounterData* method), 143
 getOverflows() (*TimeTaggerBase* method), 78
 getOverflows() (*TimeTagStreamBuffer* method), 148
 getOverflowsAndClear() (*TimeTaggerBase* method), 78
 getOverflowsAndClearClient() (*TimeTaggerNetwork* method), 95
 getOverflowsClient() (*TimeTaggerNetwork* method), 94
 getPcbVersion() (*TimeTagger* method), 85
 getPeriodsCount() (*FrequencyCounterData* method), 142
 getPeriodsFraction() (*FrequencyCounterData* method), 143
 getPhase() (*FrequencyCounterData* method), 143
 getPixelBegins() (*FlimFrameInfo* method), 133
 getPixelEnds() (*FlimFrameInfo* method), 134
 getPixelPosition() (*FlimFrameInfo* method), 133
 getPsPerClock() (*TimeTagger* method), 86
 getReadyFrame() (*Flim* method), 130
 getReadyFrameEx() (*Flim* method), 130
 getReadyFrameIntensity() (*Flim* method), 131
 getReferenceOffsets() (*PulsePerSecondData* method), 145
 getReplaySpeed() (*TimeTaggerVirtual* method), 93
 getSensorData() (*TimeTagger* method), 87
 getSerial() (*TimeTagger* method), 84
 getSignalOffsets() (*PulsePerSecondData* method), 146
 getSoftwareClockState() (*TimeTaggerBase* method), 79
 getStatus() (*PulsePerSecondData* method), 146
 getSTDD() (*FrequencyStabilityData* method), 139
 getSumChannel() (*Combinations* method), 100
 getSummedCounts() (*FlimFrameInfo* method), 133
 getSummedFrames() (*Flim* method), 131
 getSummedFramesEx() (*Flim* method), 131
 getSummedFramesIntensity() (*Flim* method), 131
 getTagger() (*SynchronizedMeasurements* method), 156
 getTau() (*FrequencyStabilityData* method), 137
 getTDEV() (*FrequencyStabilityData* method), 139
 getTestSignal() (*TimeTagger* method), 84
 getTestSignalDivider() (*TimeTagger* method), 87
 getTime() (*CounterData* method), 113
 getTime() (*FrequencyCounterData* method), 142
 getTimestamps() (*TimeTagStreamBuffer* method), 147
 getTimeTaggerChannelNumberScheme() built-in function, 73
 getTimeTaggerServerInfo() built-in function, 72
 getTotalEvents() (*FileWriter* method), 149
 getTotalSize() (*FileWriter* method), 150
 getTraceFrequency() (*FrequencyStabilityData* method), 140
 getTraceFrequencyAbsolute() (*FrequencyStabilityData* method), 140
 getTraceIndex() (*FrequencyStabilityData* method), 140
 getTracePhase() (*FrequencyStabilityData* method), 140
 getTriggerLevel() (*TimeTagger* method), 81
 getUsageStatisticsReport() built-in function, 75
 getUsageStatisticsStatus() built-in function, 74
 getUtcDates() (*PulsePerSecondData* method), 146
 getUtcSeconds() (*PulsePerSecondData* method), 146
 getVersion() built-in function, 74

H

Hardware delay, 161
 hasData() (*FileReader* method), 151
 hasOverflows (*TimeTagStreamBuffer* attribute), 147
 HIGH (*State* attribute), 153

HighResA (*Resolution attribute*), 70
HighResAll (*ChannelEdge attribute*), 69
HighResB (*Resolution attribute*), 70
HighResC (*Resolution attribute*), 70
HighResFalling (*ChannelEdge attribute*), 69
HighResRising (*ChannelEdge attribute*), 69
Histogram (*built-in class*), 116
Histogram2D (*built-in class*), 120
HistogramLogBins (*built-in class*), 118
HistogramLogBinsData (*built-in class*), 119
HistogramND (*built-in class*), 122

I

ideal_clock_channel (*SoftwareClockState attribute*), 95
initialize() (*Flim method*), 132
initialize() (*FlimBase method*), 135
injectCurrentState() (*TriggerOnCountrate method*), 106
Input time stamp, 161
input_channel (*SoftwareClockState attribute*), 95
is_locked (*SoftwareClockState attribute*), 95
isAbove() (*TriggerOnCountrate method*), 106
isAcquiring() (*Flim method*), 132
isAcquiring() (*FlimBase method*), 135
isBelow() (*TriggerOnCountrate method*), 106
isConnected() (*TimeTaggerNetwork method*), 94
isRunning() (*IteratorBase method*), 109
isRunning() (*SynchronizedMeasurements method*), 156
isServerRunning() (*TimeTagger method*), 88
isUnusedChannel() (*TimeTaggerBase method*), 80
isValid() (*FlimFrameInfo method*), 133
IteratorBase (*built-in class*), 108

L

Last (*CoincidenceTimestamp attribute*), 69
last_ideal_clock_event (*SoftwareClockState attribute*), 95
ListedFirst (*CoincidenceTimestamp attribute*), 69
Listen (*AccessMode attribute*), 68
LOW (*State attribute*), 153

M

mergeStreamFiles()
 built-in function, 73
MissedEvents (*TagType attribute*), 70
mutex (*CustomMeasurement attribute*), 157

O

Open (*GatedChannelInitial attribute*), 70
overflow (*CounterData attribute*), 112
overflow_samples (*FrequencyCounterData attribute*), 142

OverflowBegin (*TagType attribute*), 70
OverflowEnd (*TagType attribute*), 70

P

period_error (*SoftwareClockState attribute*), 95
phase_error_estimation (*SoftwareClockState attribute*), 95
pixel_position (*FlimFrameInfo attribute*), 133
pixels (*FlimFrameInfo attribute*), 132
process() (*CustomMeasurement method*), 157
PulsePerSecondData (*built-in class*), 145
PulsePerSecondMonitor (*built-in class*), 144

Q

QSFP_40GE (*FpgaLinkInterface attribute*), 70

R

ready() (*CountBetweenMarkers method*), 115
ready() (*Scope method*), 153
ready() (*TimeDifferences method*), 126
registerMeasurement() (*SynchronizedMeasurements method*), 155
replay() (*TimeTaggerVirtual method*), 92
reset() (*TimeTagger method*), 81
Resolution (*built-in class*), 70
Rising (*ChannelEdge attribute*), 69

S

sample_offset (*FrequencyCounterData attribute*), 142
Sampler (*built-in class*), 154
sampling_interval (*FrequencyCounterData attribute*), 142
scanTimeTagger()
 built-in function, 72
scanTimeTaggerServers()
 built-in function, 72
Scope (*built-in class*), 152
setConditionalFilter() (*TimeTagger method*), 81
setDeadtime() (*TimeTaggerBase method*), 77
setDelay() (*DelayedChannel method*), 102
setDelayClient() (*TimeTaggerNetwork method*), 94
setDelayHardware() (*TimeTaggerBase method*), 76
setDelaySoftware() (*TimeTaggerBase method*), 76
setEventDivider() (*TimeTagger method*), 82
setHardwareBufferSize() (*TimeTagger method*), 85
setInputDelay() (*TimeTaggerBase method*), 76
setInputHysteresis() (*TimeTagger method*), 83
setInputImpedanceHigh() (*TimeTagger method*), 82
setLED() (*TimeTagger method*), 87
setLogger()
 built-in function, 73
setMaxCounts() (*TimeDifferences method*), 125
setMaxFileSize() (*FileWriter method*), 149

setNormalization() (*TimeTagger* method), 84
 setReplaySpeed() (*TimeTaggerVirtual* method), 93
 setSoftwareClock() (*TimeTaggerBase* method), 78
 setSoundFrequency() (*TimeTagger* method), 88
 setStreamBlockSize() (*TimeTagger* method), 86
 setTestSignal() (*TimeTagger* method), 84
 setTestSignalDivider() (*TimeTagger* method), 86
 setTimeTaggerChannelNumberScheme()
 built-in function, 73
 setTimeTaggerNetworkStreamCompression()
 (*TimeTagger* method), 86
 setTriggerLevel() (*TimeTagger* method), 81
 setUsageStatisticsStatus()
 built-in function, 74
 SFPP_10GE (*FpgaLinkInterface* attribute), 70
 size (*CounterData* attribute), 112
 size (*FrequencyCounterData* attribute), 142
 size (*PulsePerSecondData* attribute), 145
 size (*TimeTagStreamBuffer* attribute), 147
 SoftwareClockState (built-in class), 95
 split() (*FileWriter* method), 149
 Standard (*Resolution* attribute), 70
 StandardAll (*ChannelEdge* attribute), 69
 StandardFalling (*ChannelEdge* attribute), 69
 StandardRising (*ChannelEdge* attribute), 69
 start() (*IteratorBase* method), 108
 start() (*SynchronizedMeasurements* method), 156
 startFor() (*IteratorBase* method), 108
 startFor() (*SynchronizedMeasurements* method), 156
 startServer() (*TimeTagger* method), 88
 StartStop (built-in class), 116
 State (built-in class), 153
 state (*Event* attribute), 153
 stop() (*Dump* method), 152
 stop() (*IteratorBase* method), 108
 stop() (*SynchronizedMeasurements* method), 156
 stop() (*TimeTaggerVirtual* method), 92
 stopServer() (*TimeTagger* method), 88
 sync() (*TimeTaggerBase* method), 79
 SynchronizedMeasurements (built-in class), 155
 SynchronousControl (*AccessMode* attribute), 68

T

TagType (built-in class), 70
 TDC time stamp, 161
 tGetData (*TimeTagStreamBuffer* attribute), 147
 time (*Event* attribute), 153
 TimeDifferences (built-in class), 125
 TimeDifferencesND (built-in class), 127
 TimeTag (*TagType* attribute), 70
 TimeTagger (built-in class), 81
 TimeTaggerBase (built-in class), 76
 TimeTaggerNetwork (built-in class), 93
 TimeTaggerVirtual (built-in class), 92

TimeTagStream (built-in class), 147
 TimeTagStreamBuffer (built-in class), 147
 triggered() (*Scope* method), 153
 TriggerOnCounterRate (built-in class), 106
 tStart (*TimeTagStreamBuffer* attribute), 147

U

UNKNOWN (*State* attribute), 153
 unregisterMeasurement() (*SynchronizedMeasurements* method), 156
 UsageStatisticsStatus (built-in class), 71

W

waitForCompletion() (*TimeTaggerVirtual* method), 92
 waitForFence() (*TimeTaggerBase* method), 79
 waitUntilFinished() (*IteratorBase* method), 109

X

xtra_getAuxOut() (*TimeTagger* method), 90
 xtra_getAuxOutSignalDutyCycle() (*TimeTagger* method), 90
 xtra_getAvgRisingFalling() (*TimeTagger* method), 89
 xtra_getClockAutoSelect() (*TimeTagger* method), 91
 xtra_getClockSource() (*TimeTagger* method), 91
 xtra_getHighPrioChannel() (*TimeTagger* method), 89
 xtra_measureTriggerLevel() (*TimeTagger* method), 91
 xtra_setAuxOut() (*TimeTagger* method), 90
 xtra_setAuxOutSignal() (*TimeTagger* method), 90
 xtra_setAvgRisingFalling() (*TimeTagger* method), 89
 xtra_setClockAutoSelect() (*TimeTagger* method), 91
 xtra_setClockOut() (*TimeTagger* method), 91
 xtra_setClockSource() (*TimeTagger* method), 91
 xtra_setHighPrioChannel() (*TimeTagger* method), 89