

Time Tagger User Manual

Release 2.18.2.0

Swabian Instruments

May 07, 2025

CONTENTS

1	Getti	ing Start	ed 1
	1.1	Get fam	iliar with your Time Tagger
	1.2	Graphic	al User Interfaces
		1.2.1	Time Tagger Lab
		1.2.2	Web Application
	1.3	Program	nming languages
		1.3.1	Python
		1.3.2	LabVIEW (via .NET)
		1.3.3	MATLAB (wrapper for .NET)
		1.3.4	Wolfram Mathematica (via .NET)
		1.3.5	.NET
		1.3.6	C#
		1.3.7	C++
2	Insta	Illation ii	ostructions 9
_	2.1	Windov	ys
		2.1.1	Installation
	2.2	Linux	9
		2.2.1	Installation
		2.2.2	Known issues
		2.2.3	Time Tagger with Python
		2.2.4	Time Tagger with C++
	2.3	Hardwa	re license upgrades
		2.3.1	Time Tagger Lab
		2.3.2	Web Application
_	_		
3	Tuto	rials	15
	3.1	Getting	started with Time Tagger Lab 15
		3.1.1	Selecting a device on the home screen
		3.1.2	Configuring input channels
		3.1.3	Performing measurements
		3.1.4	Processors and virtual channels
		3.1.5	Saving measurement data
		3.1.6	Reference clock 21
		3.1.7	Network server
		3.1.8	Troubleshooting
	3.2	Measur	ing Coincidences
		3.2.1	Time Tagger configuration 23
		3.2.2	Coincidence-counting
		3.2.3	Delay adjustment for coincidence detection

		3.2.4	Exclusive coincidences using Combinations virtual channel	26
		3.2.5	Coincidence-counting vs Correlation Measurement	28
	3.3	Confoc	al Fluorescence Microscope	29
		3.3.1	Time Tagger configuration	30
		3.3.2	Intensity scanning microscope	31
		3.3.3	Fluorescence Lifetime Microscope	32
		3.3.4	Alternative pixel trigger formats	33
	34	Ontical	ly Detected Magnetic Resonance	35
	5.1	3 4 1	Creation of optical and microwave pulse patterns	36
		342	Signal generation and detection	38
		3 4 3	Sugening modes	30
		3.4.3	ODMP contrast	10
	2 5	Domoto	Sunshronization of Time Teggers	40
	3.3		Experimentation of Time Taggers	41
		3.5.1	Establishing a common time base across distributed locations	42
		3.5.2	Starting a Time Tagger Server at each location	43
		3.5.3	Connecting to multiple Time Tagger Servers over the network	44
		3.5.4	Accessing individual servers	44
		3.5.5	Verification of the synchronization technology using a single Time Tagger	45
		3.5.6	Measuring synchronization precision across multiple Time Taggers	46
	3.6	Remote	Time Tagger with Python	47
		3.6.1	Sharing a Time Tagger with Network Time Tagger	47
		3.6.2	Remote control of a Time Tagger with Pyro	48
		3.6.3	Remote procedure call	49
		3.6.4	Initial setup	49
		3.6.5	Minimal example	49
		3.6.6	Creating the Time Tagger	50
		3.6.7	Measurements and virtual channels	51
		368	Working example	53
		0.0.0		~~
		3.6.9	What is next?	54
		3.6.9	What is next?	54
4	Sync	3.6.9 hronizer	What is next?	54 55
4	Sync 4.1	3.6.9 hronizer Overvie	What is next? . . .	54 55 55
4	Sync 4.1 4.2	3.6.9 hronizer Overvie Key ap	What is next?	54 55 55
4	Sync 4.1 4.2	3.6.9 hronizer Overvid Key apj 4.2.1	What is next?	54 55 55 55
4	Sync 4.1 4.2	3.6.9 hronizer Overvio Key ap 4.2.1 4.2.2	What is next?	54 55 55 55 55
4	Sync 4.1 4.2	3.6.9 hronizer Overvid Key app 4.2.1 4.2.2 4.2.3	What is next?	54 55 55 55 55
4	Sync 4.1 4.2	3.6.9 hronizer Overvid Key apj 4.2.1 4.2.2 4.2.3 4.2.3	What is next?	54 55 55 55 55 55 55
4	Sync : 4.1 4.2	3.6.9 hronizer Overvio Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require	What is next?	54 55 55 55 55 55 55 55 55
4	Sync 4.1 4.2 4.3	3.6.9 hronizer Overvie Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable of	What is next?	54 55 55 55 55 55 55 55 56 56
4	Sync 4.1 4.2 4.3 4.4	3.6.9 hronizer Overvie Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c	What is next?	54 55 55 55 55 55 55 55 55 56 56
4	Sync 4.1 4.2 4.3 4.4	3.6.9 hronizer Overvie Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar	What is next?	54 55 55 55 55 55 55 56 56 57
4	Sync 4.1 4.2 4.3 4.4 4.5	3.6.9 hronizer Overvic Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1	What is next?	54 55 55 55 55 55 55 55 56 56 57 58
4	Sync 4.1 4.2 4.3 4.4 4.5	3.6.9 hronizer Overvie Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2	What is next?	54 55 55 55 55 55 55 55 55 56 56 57 58 59
4	Sync 4.1 4.2 4.3 4.4 4.5	3.6.9 hronizer Overvie Key app 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2	What is next? ew plications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ements onnections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows	54 55 55 55 55 55 55 55 56 56 56 57 58 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6	3.6.9 hronizer Overvic Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat	What is next? ew plications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ements onnections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows ions	54 55 55 55 55 55 55 55 55 55 56 56 56 57 58 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6	3.6.9 hronizer Overvie Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1	What is next?	54 55 55 55 55 55 55 55 56 56 57 58 59 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6	3.6.9 hronizer Overvie Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2	What is next?	54 55 55 55 55 55 55 55 55 55 56 56 56 57 58 59 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7	3.6.9 hronizer Overvie Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable of 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I	What is next?	54 55 55 55 55 55 55 55 56 56 57 58 59 59 59 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7	3.6.9 hronizer Overvie Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I	What is next? w plications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger synchronizer with a single Time Tagger onnections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows conditional filter Internal test signal LEDs and troubleshooting	54 55 55 55 55 55 55 55 55 55 55 56 57 58 59 59 59 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7 Hard 5 1	3.6.9 hronizer Overvic Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I	What is next? www	54 55 55 55 55 55 55 55 55 55 55 55 55 5
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7 Hard 5.1	3.6.9 hronizer Overvic Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I Iware Operati	What is next? ew blications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ements connections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows ions Conditional filter Internal test signal LEDs and troubleshooting	54 55 55 55 55 55 55 55 55 55 55 55 55 5
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7 Hard 5.1 5.2	3.6.9 hronizer Overvic Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I Iware Operati Input cl	What is next? www.objections Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ements onnections Using an external reference clock e and channel numbering Incomplete cable connections Buffer overflows ions Conditional filter Internal test signal LEDs and troubleshooting	54 55 55 55 55 55 55 55 55 56 57 58 59 59 59 59 59 59 59 59 59 59
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7 Hard 5.1 5.2	3.6.9 hronizer Overvic Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I Iware Operati Input cl 5.2.1	What is next? www.initial is next? plications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ements onnections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows ions Conditional filter Internal test signal LEDs and troubleshooting Electrical characteristics	54 55 55 55 55 55 55 55 55 55 55 55 55 5
4	Sync 4.1 4.2 4.3 4.4 4.5 4.6 4.7 Hard 5.1 5.2	3.6.9 hronizer Overvie Key apj 4.2.1 4.2.2 4.2.3 4.2.4 Require Cable c 4.4.1 Softwar 4.5.1 4.5.2 Limitat 4.6.1 4.6.2 Status I Iware Operati Input cl 5.2.1 5.2.2 5.2.2 5.2.2	What is next? W Dications Crosstalk elimination High transfer rate Multi-room experiment Synchronizer with a single Time Tagger ments onnections Using an external reference clock re and channel numbering Incomplete cable connections Buffer overflows conditional filter Internal test signal LEDs and troubleshooting ng conditions Electrical characteristics Configurable input termination - Time Tagger X only	54 55 55 55 55 55 55 55 55 55 55 55 55 5

	5.3	Data connection				 			• • •	•••		63
	5.4	LEDs				 				•••		63
		5.4.1 Time Tagger X				 				•••		63
		5.4.2 Time Tagger Ultr	a			 				•••		64
		5.4.3 Time Tagger 20				 				•••		65
	5.5	Test signal				 				•••		66
	5.6	Synthetic input delay				 				•••		66
	5.7	Synthetic dead time				 						66
	5.8	Event divider				 						66
	5.9	Conditional Filter				 						66
	5.10	Bin equilibration				 						67
	5.11	Overflows				 						67
	5.12	External Clock Input				 						67
	5.13	Synchronization signals				 						68
	5.14	FPGA link				 						68
	5 1 5	General purpose IO (GPI)))			 			•••		•	68
	5.16	19-inch rack mount	,,			 	••••		•••		•	69
	5.10					 	••••		•••		•	07
6	Softw	vare Overview										71
	6.1	Graphical User Interfaces				 						71
	6.2	Precompiled libraries and	high-level langua	ge bindin	igs	 						71
	6.3	C++ API				 						71
7	Appl	ication Programming Inte	rface									73
	7.1	Examples				 				•••		73
		7.1.1 Measuring cross-	correlation			 				•••		73
		7.1.2 Using virtual char	nnels			 				•••		74
		7.1.3 Using multiple Ti	me Taggers			 				•••		74
		7.1.4 Using Time Tagg	er remotely			 				•••		74
	7.2	The TimeTagger Library				 				•••		75
		7.2.1 Units of measurer	nent			 				•••		75
		7.2.2 Channel numbers				 				•••		76
		7.2.3 Unused channels				 				•••		76
		7.2.4 Constants				 				•••		76
		7.2.5 Enumerations .				 						76
		7.2.6 Functions				 						80
		7.2.7 Helper classes .				 						84
	7.3	TimeTagger classes				 						84
		7.3.1 General Time Tag	ger features			 				•••		85
		7.3.2 Time Tagger				 				•••		93
		7.3.3 The TimeTagger	/irtual class			 				•••		104
		7.3.4 The TimeTagger	Vetwork class			 						106
		7.3.5 Additional classes	3			 						109
	7.4	Virtual Channels				 						111
		7.4.1 Available virtual	channels			 						111
		7.4.2 Common method	s			 						112
		7.4.3 Coincidence				 						112
		7.4.4 Coincidences				 						113
		7.4.5 Combinations	· · · · · · · · · · · · ·			 						115
		7.4.6 Combiner				 						117
		7.4.7 ConstantFraction	Discriminator			 						117
		7.4.8 DelavedChannel				 						119
		7.4.9 EventGenerator				 						119
		7.4.10 FrequencyMultin	lier			 						120
		1 P				 		•				

		7.4.11	GatedChannel	121
		7.4.12	TriggerOnCountrate	122
	7.5	Measur	rement Classes	124
		7.5.1	Available measurement classes	124
		7.5.2	Common methods	125
		7.5.3	Event counting	127
		7.5.4	Time histograms	132
		7.5.5	Fluorescence-lifetime imaging (FLIM)	143
		756	Phase & frequency analysis	151
		757	Time-tag-streaming	163
		758	Helper classes	172
		7.5.0		1/2
8	In De	epth Gui	ides	175
Ŭ	81	Softwar	re-Defined Reference Clock	175
	0.1	811	Overview of synchronization concepts	175
		812	Setting up the software-defined reference clock	176
		0.1.2 Q 1 2	Technical limitations	176
		0.1.3	Advanced features	170
	0.0	0.1.4 Canditi		179
	8.2			1/9
		8.2.1		181
		8.2.2	Understanding the filtering mechanism	183
		8.2.3	Setup of the Conditional Filter	185
	8.3	Raw Ti	me-Tag-Stream access	186
		8.3.1	Dumping and post-processing	186
		8.3.2	On-the-fly processing	187
	8.4	Synchro	onization of the Time Tagger pipeline	187
	8.5	FPGA 1	link	188
		8.5.1	Getting Started with SFP+	188
		8.5.2	Using QSFP+	189
		8.5.3	Modifying the reference design	189
9	Usag	e Statisti	ics Collection	191
	9.1	Content	ts of the usage statistics data	191
	9.2	Ways of	f control	191
	9.3	Time Ta	agger Lab diagnostics	192
10	Safet	y & Con	npliance	193
	10.1	Safety a	and compliance guidelines	193
		10.1.1	Symbols	193
		10.1.2	Operation environment	193
		10.1.3	Electrostatic-sensitive device	193
		10.1.4	Disposal and recycling	194
		10.1.5	Contact, support and service	194
	10.2	Time Ta	agger X safety notice	194
		10.2.1	Electrical characteristics	195
		10.2.2	Equipment installation	195
		10.2.3	Maintenance and renair	196
		10.2.5		170
11	Revis	sion Hist	torv	199
	11.1	V2.18.2	2 - 07.05.2025	199
	11.2	V2.18 (0 - 23.04.2025	199
	11.3	V2 17 6	6 - 21.01.2025	201
	11.5	V2.17.4	4 - 17 07 2024	201
	11.5	V2 17 0	2 - 02 07 2024	201
	11.5	V2.17.2	$2 02.07.202 + \dots + $	201
				2.3.1

11.7	V2.16.2 - 28.06.2023				 			 203						
11.8	V2.16.0 - 05.06.2023				 			 203						
11.9	V2.15.0 - 06.03.2023				 			 204						
11.10	V2.14.0 - 23.12.2022				 			 205						
11.11	V2.13.2 - 22.11.2022				 			 205						
11.12	V2.12.4 - 09.11.2022				 			 206						
11.13	V2.12.2 - 04.10.2022				 			 206						
11.14	V2.12.0 - 01.09.2022				 			 206						
11.15	V2.11.0 - 22.04.2022				 			 207						
11.16	V2.10.6 - 16.03.2022				 			 208						
11.17	V2.10.4 - 23.02.2022				 			 209						
11.18	V2.10.2 - 31.12.2021				 			 209						
11.19	V2.10.0 - 22.12.2021				 			 209						
11.20	V2.9.0 - 07.06.2021				 			 211						
11.21	V2.8.4 - 04.05.2021				 			 211						
11.22	V2.8.2 - 26.04.2021				 			 211						
11.23	V2.8.0 - 29.03.2021				 			 212						
11.24	V2.7.6 - 26.04.2021				 			 213						
11.25	V2.7.4 - 19.04.2021				 			 213						
11.26	V2.7.2 - 22.12.2020				 			 213						
11.27	V2.7.0 - 01.10.2020				 			 214						
11.28	V2.6.10 - 07.09.2020				 			 214						
11.29	V2.6.8 - 21.08.2020				 			 215						
11.30	V2.6.6 - 10.07.2020				 			 215						
11.31	V2.6.4 - 27.05.2020				 			 216						
11.32	V2.6.2 - 10.03.2020				 			 217						
11.33	V2.6.0 - 23.12.2019				 			 218						
11.34	V2.4.4 - 29.07.2019				 			 220						
11.35	V2.4.2 - 12.05.2019				 			 220						
11.36	V2.4.0 - 10.04.2019				 			 220						
11.37	V2.2.4 - 29.01.2019				 			 221						
11.38	V2.2.2 - 13.11.2018				 			 221						
11.39	V2.2.0 - 07.11.2018				 			 221						
11.40	V2.1.6 - 17.05.2018				 			 222						
11.41	V2.1.4 - 21.03.2018				 			 222						
11.42	V2.1.2 - 14.03.2018				 			 222						
11.43	V2.1.0 - 06.03.2018				 			 222						
11.44	V2.0.4 - 01.02.2018				 			 222						
11.45	V2.0.2 - 17.01.2018				 			 222						
11.46	V2.0.0 - 14.12.2017				 			 223						
11.47	V1.0.20 - 24.10.2017				 			 223						
11.48	V1.0.6 - 16.03.2017				 			 224						
11.49	V1.0.4 - 24.11.2016				 			 224						
11.50	V1.0.2 - 28.07.2016				 			 225						
11.51	V1.0.0				 			 225						
11.52	Channel Number Sche	ema	0 an	d 1	 		 •	 225						

Index

CHAPTER

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, and MATLAB.

 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 folders of your Time Tagger installation.

1.2 Graphical User Interfaces

1.2.1 Time Tagger Lab

Time Tagger Lab is the Graphical User Interface (GUI) application for Windows operating systems. It is designed to perform standard measurements quickly and to get an interactive experience with your Time Tagger. Here, you can read a brief description how you can set up a simple *cross correlation* measurement in Time Tagger Lab. For a full tutorial to the software, please refer to *getting started with Time Tagger Lab*.

- 1. Download and install the most recent Time Tagger software from our downloads site.
- 2. Start Time Tagger Lab from the Windows start menu.



3. The Time Tagger Lab welcome screen opens and presents all available devices. If you plug or unplug a Time Tagger, the display will update automatically.

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 *l* 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.

τ	Time 1	agger Lab	v2.17.4 — 🗆 🗙
	Time Tagger Ullra	Swabian instruments	
	2250001398 High-Res		
	Use default workspace CAUsersVanBruin\Documents\Time Reset workspace		

- 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 the Time Tagger software is installed.
- 2. Make sure that your *Time Tagger* device is connected to your computer and the *Time Tagger Lab* is closed.
- 3. Make sure a Python distribution is available.
 - 1. On Unix/Linux systems, a system-supported installation of Python should already be present.
 - 2. On **Windows** systems, you can install Python by following these instructions. We recommend the default installation ("Install Now"), as it includes all standard libraries and pip.
- 4. Make sure to install the required Python packages NumPy, Matplotlib, and ipython.
- 5. Open a command shell and navigate to the .\examples\python\1-Quickstart folder in your *Time Tagger* installation directory.
- 6. Start an **ipython** shell with plotting support by entering **ipython** --pylab.
- 7. 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.	Торіс	Classes & Methods
Basic software con	trol (folder 1-basic_soj	ftware_control)
1-A	Create a measure- ment	<pre>createTimeTagger(), Counter::getData(), Counter::getIndex()</pre>
	Count rate trace	Counter
1-B	Start & stop mea- surements	<pre>Countrate, start(), stop(), startFor()</pre>
1-C	Synchronize mea- surements	SynchronizedMeasurements
	Use different his-	Correlation, Histogram,
	tograms	StartStop,HistogramLogBins
1-D	Virtual Channels	DelayedChannel, Coincidence, GatedChannel
1-E	Logging errors	setLogger()
1-F	External reference	TimeTaggerSource::setReferenceClock(),
	clock	FrequencyStability
Controlling the ha	rdware (folder 2-control	olling-the-hardware)
2-A	Get hardware infor-	<pre>scanTimeTagger(), getSerial(),</pre>
	mation	<pre>getModel(), getSensorData(),</pre>
		getConfiguration()
2-B	The input trigger level	<pre>setTriggerLevel(), getTriggerLevelRange()</pre>
2-C	Filter tags on hard-	<pre>setConditionalFilter(),</pre>
	ware	setEventDivider()
2-D	Control input delays	setInputDelay(),Histogram2D
2-E	Overflows	<pre>getOverflows(),</pre>
		<pre>setTestSignalDivider()</pre>
2-F	HighRes mode	<pre>createTimeTagger(), TimeDifferences</pre>
Dump and re-anal	yze time-tags (folder 3	-dump-and-reanalyze-time-tags)
3-A	Dump tags by FileWriter	FileWriter
3-В	The Time Tagger Virtual	<pre>createTimeTaggerVirtual(), TimeTaggerVirtual</pre>
Working with raw	time-tags (folder 4-wo	rking-with-raw-time-tags)
4-A	The FileReader	FileReader, TimeTagStreamBuffer
4-B	Streaming raw time-tags	TimeTagStream
4-C	Custom Measure- ments	CustomMeasurement

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).

Untitled 1 Block Diagram	– D X	Untitled 1 Block Diagram —	
File Edit View Project Operate Tools Window Help		File Edit View Project Operate Tools Window Help	1000
💠 🕸 🥚 🛚 🎯 👷 မဝ 🔂 ာ TSpt Application Font 🔻 ဦာနာ 🙃 န 🔅 နောက်	् 🤋 🚟	🐴 🐵 🕘 🖩 🌚 👷 🦏 🗤 🗃 🗗 15pt Application Font 🔻 🏪 🛪 🏟 💘 🔩 • Search	् १ 🗠
Functions Q. Search Customize* Image: Customize* </th <th></th> <th>Select NET Constructor X Swabianinstruments.TimeTagger(2.16.2.0) v Browse Digets Swabianinstruments.TimeTagger(2.16.2.0) v Browse October Ostantificage Sindidence Ostindence Sindidence Sindidence Ostindence Sindidence Sindidence Ostindence Sindidence Sindidence Ostindence Constructors Constructors OK Cancel Help</th> <th></th>		Select NET Constructor X Swabianinstruments.TimeTagger(2.16.2.0) v Browse Digets Swabianinstruments.TimeTagger(2.16.2.0) v Browse October Ostantificage Sindidence Ostindence Sindidence Sindidence Ostindence Sindidence Sindidence Ostindence Sindidence Sindidence Ostindence Constructors Constructors OK Cancel Help	

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.

Several examples demonstrating how to use the Time Tagger with MATLAB are provided in the .\examples\Matlab\ folder. The 1-Quickstart directory is structured into three subfolders (1-, 2-, and 3-), mirroring the Python quickstart examples. These cover core topics such as basic measurements, hardware control, and postprocessing, providing a one-to-one correspondence between MATLAB and Python usage.

1.3.4 Wolfram Mathematica (via .NET)

Time Tagger functionality is provided to Mathematica via the .NET interoperability interface.

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.

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

The choice of build type can have a great effect on the code performance. In Visual Studio, the default compiler flags for Debug builds are /MDd, /00, and /Zi. For Release builds these are /MD, /02, and /DNDEBUG. It is crucial that you understand these flags and their implications on performance.

• /00 vs /02

This flag controls the general optimization level. /00 means no optimization, so every instruction is surrounded by load_from_memory and store_to_memory. This is a significant waste of CPU resources, but guarantees that every local variable can be inspected at all times. We suggest the use of the default op-timization level /02 and to overwrite it only for required methods using pragmas (see the 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 using /MD even for debug builds, otherwise

the debug runtime will incur significant performance costs, including in the internals of the Time Tagger library.

• /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, we recommend using this flag for all internal builds.

• /DNDEBUG

The macro NDEBUG is the C way to disable assert(). Asserts are usually good at catching logic errors without much overhead.

The relevant choice is thus not whether to build using Debug or Release, but rather which set of flags to use. Please select each of them carefully based on your target.

Code Sanitizers

We recommend the use of code sanitizers. Switching your Visual Studio compiler to CLANG allows you to use its undefined behavior, address, and thread sanitizers. Microsoft has also recently announced the MSVC compiler now ships with an address sanitizer, which finds out-of-bounds accesses, use-after-free and similar issues. Please consult the documentation for the AddressSanitizer (ASan) for Windows with MSVC.

CHAPTER

INSTALLATION INSTRUCTIONS

2.1 Windows

Time Tagger software requires Windows 10 or higher (64 bit). For Windows 10, we provide full support only for those versions that are still actively supported by Microsoft. Older versions, while untested, might still work.

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 (-1 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.

Note

It is currently not possible to update the hardware licenses of Time Taggers that are connected to a *Synchronizer* using Time Tagger Lab. To perform a license update, power off the Synchronizer first by unplugging its power supply. Then, the Time Taggers will be listed one-by-one and their licenses can be upgraded as described in the following section.

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:

	Swabian instruments
Time Tagger X	Time Tagger Virtual
23270014ZB Performance	
	$\bigcirc \bigcirc \cdot \cdot \cdot \cdot \odot \bigcirc$
High-Res 2 ps	
Force high impedance for all channels	
New device license is available <u>Update the license</u>	

	Swabian instruments
Time Tagger X	Time Tagger Virtual
23270014ZB High-Res	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	© © · · · ⊙ ⊙
High-Res 2 ps	
Force high impedance for all channels	
License successfully updated	

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)

ŀ	Hardware Device scanner
	▲ Feature development for the Web Application is discontinued. Please use Time Tagger Lab for new projects. Time Tagger Lab is part of the Time Tagger software installer so it is already available on your PC.
	1 Hardware Device(s) found!
	Serial: 2223001154 Model: Time Tagger X ★ Init with resolution: Standard Status: ✓ Ready
	Demo device: TimeTagger Virtual Control Init
	Q Scan again
	Check for License Upgrade

- 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:

License Upgrade
Serial: 2223001154 ID: 98467210993894020 Model: Time Tagger X Current License: Edition: HighRes Channels: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17] Date: 2023-03-03T08:41:202 New license available: Edition: HighRes Channels: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18] Status: \uparrow New License is available. Please flash your device. IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
× Close

License Upgrade	
Serial: 2223001154 ID: 98467210993894020 Model: Time Tagger X Current License: Edition: HighRes Channels: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18] Date: 2023-03-03T09:32:08Z	
Status: ✔ Time Tagger was successfully flashed	
	X Close

CHAPTER

THREE

TUTORIALS

3.1 Getting started with Time Tagger Lab

Time Tagger Lab is the Windows GUI for Swabian Instruments' Time Tagger devices that you can use to run measurements with just a few clicks. Information on how to download and install the software and how to initiate a basic testing experiment can be found in the *Getting Started* section of the documentation. The purpose of this tutorial is to explore the capabilities of Time Tagger Lab in more detail and to cover the most common aspects of configuring and running measurements with it.

3.1.1 Selecting a device on the home screen

On the home screen of Time Tagger lab, you will see all Time Tagger devices that are attached to your computer. Each device can be identified by its serial number, shown in the top left of the corresponding card. If your Time Tagger is not listed here, it may be because it is in use by another program or not yet powered up.

τ	Time Ta	gger Lab	v2.17.4 — 🗆 🗙
 Time 1	agger Ulfra	Swabian instruments	
22500013	98 High-Res		
	Use default workspace CAUsers\JanBruin\Documents\Time T		

Note

Time Tagger Ultra and *Time Tagger X* have a "HighRes" option to achieve improved jitter that can be updated over-the-air. If you want to operate in these modes, you should select them here before clicking on the card. If you are interested in this option and your device has not been configured appropriately, please contact our team at sales@swabianinstruments.com.

For the *Time Tagger X* only, there is an additional option to select input impedance, see: setInputImpedanceHigh().

At the bottom of the screen, the workspace menu allows you to select the folder in which your configuration and settings will be stored, as well as an option to overwrite your previous workspace to default values.

The card named Time Tagger Virtual allows you to experiment with the functionality of Time Tagger Lab without a physical Time Tagger, using only simulated signals.

3.1.2 Configuring input channels

The first window that is shown is the device view. Here, you see a graphical representation of your Time Tagger with the available channels highlighted. The live count rate and trigger level are shown above and below each input, respectively. The trigger level can be adjusted with a slider that appears when hovering over the value. If you are using a *Synchronizer*, you can cycle through all synchronized Time Taggers using the arrow buttons to the right and left of the device.

Below the device view, you will see status indicators for the reference software clock and the time tagger server. To access detailed options for device settings, and to configure the reference clock and/or server, use the slider in the top left corner to switch to Detailed View.

T File View Windows Help		Time Tagger Lab		AAA • v2.17.7-45-gaa5f0e92 🗕 🗖 🗙
Ů ⊡ ⊟ 88 ⊡ ⊡ ⊡ Ξ			(Measurements running: 0 Feedback
Measurements v 4				
Quick add measurement Open creator (F2)	Detailed View			
Processors	0 (1) 0.500 V	2 3 4 5 assov	6 7 8 9 0.500 v	cps f
				•
		Time Tagger Ull		•
				•
	0 10 0.500 V	(1) (2) (3) (0) asoov	15 (16 (17 (18) 0.500 V	cps f
	2411001AA	A		LEDs are enabled
	Reference software clock OFF		Time Tagger server OFF	
	Channel:		Access mode: Listen 👻	Channels:

In the detailed view, all channel configurations are listed in a table. Most parameters may be applied separately to rising and falling edges and all channels are treated fully independently of one another. Optionally, channels can be

assigned a name to make them easier to identify later. When tuning the channel parameters (e.g. *Trigger level*, *Signal delay*, *Dead time*, *Event divider*, *Conditional filter*), it can be very useful to monitor their influence on the displayed live count rate. These values are shown in a separate column near the right hand side of the table, in counts per second ("cps", rising edge events only).

T File View Windows Help	Time Tagger Lab TTU 2411001A4						\$11001AAA	• v2.17.7	-45-gaa5	f0e92 —	Ξ×				
└ └ 世 88 └ □ 🗉 Ξ											•	Measurem	ents runn	ing: 0	Feedback
Measurements v 🐺															
Quick add measurement Open creator (F2)	<u> </u>	etailed View	N												
nents	#	Signal delay Dead time			Event	Event divider Conditional filter					Countrate	Test			
Proces		Hame	inggeniet							τÈ	ł۲	₹F	łF	(cps)	signal
102			0.500 V	0 ps	0 ps	2 ns	2 ns			D	D	D			
			0.500 V	0 ps	0 ps	2 ns	2 ns			O	o				o
			0.500 V	0 ps	0 ps	2 ns	2 ns				0				
			0.500 V	0 ps	0 ps	2 ns	2 ns				O				
			0.500 V	0 ps	0 ps	2 ns	2 ns			0	0	O			D
			0.500 V	0 ps	0 ps	2 ns	2 ns			0	0				D
		Poforone	n software slov	L OFF											
	Chan	nel:		K OIT				Referer	ice frequency	(Hz): 10	000 000				
	Avera	iging period	ds: 1000					Input fr	equency (Hz						
	Access	mode: Lis	ten					Channel	ls:						
	Port	41	101												

Trigger level

Adjusting the trigger level can help to maximize the signal to noise ratio of your measurement. You can adjust the trigger level in 1 mV increments, see: *setTriggerLevel()*. The available range of trigger levels depends on the specific Time Tagger model, as listed in the *Hardware* section. Often, the trigger level is set around half the maximum amplitude of the signal, as this is where the signal tends to be steepest. Please refer to our knowledge base article on trigger levels for more details.

Signal delay

You can add an arbitrary delay to the signal with 1 ps resolution, see: *setInputDelay()*. The delay may be either positive or negative. This setting can be used to offset or align signals in time at different inputs, e.g. to correct for different cable lengths. For a more detailed discussion on the use of software and hardware delays, please refer to our knowledge base article on delay compensation.

Dead time

Because the dead time of the Time Tagger is often shorter than that of a detector, undesired signals (e.g. from afterpulsing) may sometimes be acquired as counts. To prevent this, one can modify the dead time to each channel individually, see: *setDeadtime()*. The default value is the minimum dead time for the Time Tagger device model.

Event divider

When working with periodic signals with very high repetition rates (e.g. clocks or pulsed lasers), it may be necessary to reduce the rate of tags to prevent *Overflows*. You can apply an event divider to the signal to divide down the rate of the signal by a factor of *n*, see: *setEventDivider()*.

Conditional filter

The conditional filter can be configured by selecting one or more *triggered* (T) and one or more *filtered* (F) channels, see: *setConditionalFilter()* as well as the in-depth guide *Conditional Filter*.

3.1.3 Performing measurements

Measurements are listed on the left-hand panel of Time Tagger Lab. To start a new measurement, either type the name of that measurement in the *Quick add a measurement* box, or open the wizard with the button Open Creator or by pressing F2. The creator wizard lists all available measurements together with their descriptions and explanatory figures. For a more detailed description of all supported measurements, please refer to the *Measurement Classes* section of the documentation.



In the figure above, a *Bidirectional Histogram* measurement is selected within the creator wizard. This measurement is typically used for performing second order correlation / photon antibunching experiments. After adjusting the input parameters, the measurement can be added with the button Add measurement. If any parameter needs to be adjusted later, it can be done easily after the measurement has started and whilst it is running. If the input configuration is incomplete for a valid measurement, a warning will be displayed.

The measurement view

Once a measurement is added, it will appear as an item in the measurements list and the corresponding chart will appear under a new tab in the main window. The main measurement controls start, stop and restart can be accessed in either location.

By default, measurement charts are organized as a horizontal list of tabs. However, you can freely rearrange them by right-clicking on a tab and choosing float or New Horizontal / Vertical Document Group. In particular on high-resolution monitors, this allows for displaying many measurements side-by-side with great flexibility.



Measurements list

The measurements list on the top left shows all configured measurements. Their corresponding charts can be hidden or shown by clicking the 'eye' button. Right-clicking on a measurement opens a menu to clone or delete it.

Chart view

The chart view on the right is a live display of the measurement data. Usually, it consists of two elements: a main chart above and an auxiliary chart below. Left of the main chart are the measurement controls, as well as zooming, panning, crosshair, data marker and logarithmic axis controls. Zooming is performed by clicking and dragging a box over the chart. Double-click on the chart to revert to automatic zoom limits. The measurement control for exporting data is described in more detail under *Saving measurement data*.

Clicking on a legend checkbox toggles the visibility of that data series, and right-clicking the legend allows it to be moved to a different corner of the chart.

For most measurements, the auxiliary chart displays either the concurrent *Counter* or *FrequencyCounter* measurement of the measured input channels, which can be useful for monitoring purposes. Switch between these two measurements by clicking on the y-axis label of the auxiliary chart.

Properties

Time Tagger Lab allows for on-the-fly changes to the inputs of your measurements in the corresponding properties box in the bottom-left corner of the screen. Changing *Display* properties only affects how the chart displays the data and does not interrupt the measurement. Changing other properties typically causes the measurement to restart, with the new settings in place. The *Capture settings* determine whether the measurement will run for a fixed amount of time (Single), repeatedly for a fixed time (Repeating) or indefinitely (Free running).

3.1.4 Processors and virtual channels

Virtual Channels have been partially implemented in Time Tagger Lab since software version 2.18.0. They are software-defined channels which behave like physical measurement channels, but are created by processors such as *Coincidences*, *GatedChannel* or *DelayedChannel*. Processors are created similarly to measurements, the main difference being that their output is not a chart but a (named) virtual channel which can be used as an input for measurements or other virtual channels. The icons used to represent physical channels are numbers within golden circles, whereas processors are represented by white hexagons around a stylized *AND* gate.



The tab for creating virtual channels is shown on the left-hand side of the screen and can be switched with the measurements tab. If it is convenient to show both the virtual channels and measurements side by side, the view can be switched with the layout buttons in the view menu. After a virtual channel is created, it will be listed under every Select/Add a channel control, alongside the physical input channels.

Processor tips and tricks

Processors can be a very powerful tool to analyze your data, but using them may involve complicated configuration interdependencies. To help prevent broken dependencies and other pitfalls, Time Tagger Lab restricts some of the choices you can make when creating and deleting processors. When editing the configuration of a processor, any measurements that use its virtual channels as inputs must be paused. In addition, the inputs to that processor may only include virtual channels belonging to previously created processors. When deleting processors or virtual channels, you will receive a warning listing all the dependent processors and measurements whose configuration will change due to the removal of the object.

When configuring multiple *Coincidences* channels with the same coincidence window, it is generally recommended to combine them in the same processor. The reason for this is that the computational performance is much higher than for many single-channel *Coincidences* processors.

Virtual channel naming

All processors must have a unique name, and Time Tagger Lab assigns a default one which you may overwrite freely to make it more descriptive. For processors with only a single output, such as *DelayedChannel*, the virtual channel name matches the processor name. For processors with multiple possible outputs, such as *Coincidences*, each virtual channel must have its own name. Again, Time Tagger Lab will assign default names which you are free to change. When selecting these virtual channels as inputs to a measurement or another processor, the channel name is selected from a drop-down menu.

New Coincide	ences Processor	+ Channel group
Properties		
Window	1 ns	
Channel group	1, 5	X -
🔵 1: Channel	<u>t</u>	≠×
🔵 2: Channel	5 ₹	≠×
Add channel		•
Channel group	My custom channel name	X
🔵 1: Channel	10 5	≒×
🔵 2: Channel	(14) ∱	≠×
Add channel		•

3.1.5 Saving measurement data

These are the two ways of saving measurement data to file within Time Tagger Lab using the relevant chart control.

Note

If you want to save raw time tags to file from Time Tagger Lab use a *Time tag file writer* measurement instead, see also: *FileWriter*.

Export data / trace data

Export data / trace data saves either the main or auxiliary chart data series as a tab-separated text file. The first line of the file contains descriptive header data.

Export chart

Save chart as saves the main measurement chart as a PNG image.

3.1.6 Reference clock

The reference clock can be configured within the detailed device view. The reference clock provides a powerful software-defined tool to rescale the time base of the measured tags to an external clock. Select the input to be used for the clock channel, input the clock frequency and start and stop the reference clock using the toggle button.

3.1.7 Network server

From the detailed device view, a network server can be configured and enabled. For a detailed discussion of Time Tagger Network, see *The TimeTaggerNetwork class*. Select the appropriate server *AccessMode*: Listen, Control, or SynchronousControl. Start and stop the server using the toggle button.

Refere	nce software clock OFF		
Channel:	Ł ()	Reference frequency (Hz):	10 000 000
Averaging per	iods: 1000	Input frequency (Hz):	
Time			
Access mode:	Listen	Channels:	
Port:	41 101		

3.1.8 Troubleshooting

Time Tagger Lab has several ways of displaying errors. Typically, these are not critical and a normal part of using the hardware. Understanding error messages helps to troubleshoot possible issues with your measurements.

Note

To reach out to Swabian Instruments user support, please contact us at support@swabianinstruments.com

Overflows

A common error is *Overflows*, which occur when the tag rate is persistently above the hardware limits. For reference, the USB 2.0 interface of the *Time Tagger 20* is capable of 9 MTags/s, and the USB 3.0 interface of the *Time Tagger Ultra* and *Time Tagger X* can perform at 90 MTags/s. However, the actual rate you can achieve also depends on the CPU of the PC the Time Tagger is attached to. For a more detailed discussion of this topic, please refer to our knowledge base article on rate limits.

To avoid overflows, consider the use of *Event divider* or *Conditional filter* to reduce the streamed data rate.

Log warning messages

Time Tagger Lab reports warning messages from the backend engine in the log viewer. Example messages would be a Time Tagger USB disconnection event, or a network client connecting to the local server. If the log viewer is not currently visible, you can open it with the menu item Show Log Viewer under Windows. Right-clicking anywhere within the log viewer gives the option to clear the log, or to suppress all existing alarms.

3.2 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 interference 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.

3.2.1 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)
```

3.2.2 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()
```

3.2.3 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, ◊)
# 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.

3.2.4 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 $\{7, 8\}$.

(continues on next page)

(continued from previous page)

```
# Create a list with the virtual channels monitoring the combinations
virtualchannelsnumber = []
for group in filtered_combinations:
    virtualchannelsnumber.append(combinations_vchannels.getChannel(group))
```

3.2.5 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 "*Coincidence* + *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 coincidinces 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.

3.3 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.

3.3.1 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 V-amplitude, we set the trigger level to 0.1 V on this channel. The default trigger level is 0.5 V.

```
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.

3.3.2 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_END_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)
```

3.3.3 Fluorescence Lifetime Microscope

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

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

```
nx_pix = 300  # Number of pixels along x-axis
ny_pix = 200  # Number of pixels along y-axis
binwidth = 50  # in picoseconds
n_bins = 2000  # number of bins in a histogram
n_pixels = nx_pix * ny_pix # number of histograms
flim = TimeDifferences(
    tt,
    click_channel=SPD_CH,
    start_channel=LASER_CH,
    next_channel=PIXEL_START_CH,
    binwidth=binwidth,
    n_bins=n_bins,
    n_histograms=n_pixels
)
```

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

```
while scanner.isScanning():
    counts = flim.getData()
    img3D = np.reshape(counts, n_bins, nx_pix, ny_pix) # Fluorescence image cube
    # User defined function that estimates fluorescence lifetime for every pixel
    flimg = get_lifetime(img3D)
    plt.imshow(flimg)
    plt.pause(0.5)
```

3.3.4 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.



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
                            # Number of pixels/lines along y-axis
ny_pix = 200
                           # Integration time of 3 ms in picoseconds
integr_time = int(3e9)
line_duration = 1e12
                            # Duration of the line scan in picoseconds
binwidth = 50
                            # in picoseconds
n_bins = 2000
                           # number of bins in a histogram
n_pixels = nx_pix * ny_pix # number of histograms
LINE\_START\_CH = 3
# Pixels are equally spaced in time (constant speed)
pixel_start_times = numpy.linspace(0, line_duration, nx_pix, dtype='int64')
# Pixel integration time is constant
pixel_stop_times = pixel_start_times + integr_time
# Create EventGenerator channels
pixel_start_vch = EventGenerator(tt, LINE_START_CH, pixel_start_times.tolist())
pixel_stop_vch = EventGenerator(tt, LINE_START_CH, pixel_stop_times.tolist())
PIXEL_START_CH = pixel_start_vch.getChannel()
PIXEL_END_CH = pixel_stop_vch.getChannel()
# Use GatedChannel to gate the detector
gated_vch = GatedChannel(tt, SPD_CH, PIXEL_START_CH, PIXEL_END_CH)
GATED_SPD_CH = gated_vch.getChannel()
flim = TimeDifferences(
   tt,
    click_channel=GATED_SPD_CH,
    start_channel=LASER_CH,
   next channel=PIXEL START CH.
   binwidth=binwidth,
   n_bins=n_bins,
   n_histograms=n_pixels
)
```

Note

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

3.4 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, as also shown in this application note.

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.



3.4.1 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,
\rightarrowLOW).
                (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
seq.plot()
```

3.4.2 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
```

3.4.3 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.

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.

```
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)
   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)
```

3.4.4 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 repated 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 *CountBetweenMarkers* 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 *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)
```

3.5 Remote Synchronization of Time Taggers

In high-precision applications such as distributed time monitoring, telecommunications, quantum communication, and quantum key distribution, synchronizing time-tagging devices across remote locations is crucial. When Time Taggers operate independently, their clocks run free, and even small differences in their frequencies accumulate over time, creating significant timing discrepancies across multi-device setups.

To overcome this challenge, a synchronization-agnostic and versatile approach is employed. This method supports any synchronization technology that provides a frequency reference signal (e.g., 10 MHz) and a 1PPS (Pulse Per Second) timing signal, as shown in the figure below. For example, the White Rabbit is a widely adopted technology that enables

picosecond-level synchronization precision with sub-nanosecond accuracy across remote setups, as detailed in this application note. The synchronization process relies on the two timing signals combined with software-based solutions, including functionalities such as the ReferenceClock, to unify the time bases of the distributed Time Taggers. Users must ensure that their hardware infrastructure supports the required synchronization signals to enable this functionality. Moreover, when using the ReferenceClock with *Time Tagger 20*, an inherent timing error of approximately 200 ps needs to be considered. More details can be found in the *setReferenceClock()* documentation and the related in-depth guide (*Software-Defined Reference Clock*).

This tutorial demonstrates how to achieve remote synchronized operation of Time Taggers. It provides a step-by-step guide to synchronizing multiple Time Taggers across distributed locations, unifying their time bases, and merging time tag streams from remote network nodes for real-time processing using the *TimeTaggerNetwork* functionality, which supports simultaneous connections to multiple servers from version 2.18 onward.



3.5.1 Establishing a common time base across distributed locations

A critical step in synchronizing Time Taggers across remote locations is establishing a unified time base. Within our approach, this is achieved by associating the 1PPS signal fed into the Time Tagger, which is generated at each location with an external clock locked to a Grand Master (GM), with the Coordinated Universal Time (UTC) provided by the host PC. To ensure accurate synchronization, the PC clocks at all locations must remain aligned to UTC with a time shift of less than 0.5 seconds. Synchronizing the PC clock is therefore essential because it allows the system to associate the 1PPS signal at each location with the correct UTC second, ensuring that all timing data is unified under a common temporal framework.

The PC clock alignment to UTC can be achieved using reliable synchronization protocols such as Network Time Protocol (NTP) or Precision Time Protocol (PTP). Standard operating system synchronization tools (e.g., Windows Time Service, Linux's *ntpd* or *chrony*) configured to synchronize with either publicly available or internal time servers, are sufficient to maintain synchronization shift below 0.5 seconds. For enhanced robustness, ease of use, or advanced configuration options, dedicated synchronization software solutions such as the Meinberg NTP software package are recommended.

3.5.2 Starting a Time Tagger Server at each location

Once the PC clocks at all locations are aligned to UTC, the next step is to configure each Time Tagger to operate within a unified time base. By default, timestamps generated by a Time Tagger are relative to when the device was initialized in software. This means that if multiple Time Taggers start at different times, identical physical events occurring simultaneously at different locations receive different timestamps.

To synchronize the Time Taggers, enabling measurements across time tag streams generated at remote locations, the ReferenceClock must be activated. This ensures that all Time Taggers apply a UTC-based offset to their time base, allowing simultaneous events at remote locations to receive identical timestamps, regardless of when each Time Tagger was started. The ReferenceClock achieves this by locking the internal clock to an external distributed frequency reference (e.g., 10 MHz) and using the 1PPS signal to establish absolute time alignment.

To set up each Time Tagger, a connection to the device is first established in software. Hardware settings such as trigger levels and dead time should be configured according to the specific measurement requirements. Then, the ReferenceClock is enabled, aligning the internal time base with the external synchronization signals. Finally, a server is started on the host PC, making the Time Tagger accessible over the network for real-time data collection.

```
import TimeTagger
# Connect to the Time Tagger via USB
tagger = TimeTagger.createTimeTagger()
# Declare the channel names and corresponding physical connections
frequency_channel = 1
PPS_channel = 2
# Define the hardware settings here, such as trigger level or dead time.
# Enable the ReferenceClock
tagger.setReferenceClock(clock_channel=frequency_channel,
                         clock_frequency=10e6,
                         time_constant = 1e-4,
                         synchronization_channel=PPS_channel,
                         wait_until_locked=true)
# Start the Server.
# TimeTagger.AccessMode sets the access rights for clients.
# Port defines the network port to be used
tagger.startServer(access_mode=TimeTagger.AccessMode.Control,port=41101)
```

Warning

Activating the ReferenceClock shifts and rescales the internal time base of the Time Tagger. As a result, all ongoing measurements on every channel will be affected. Data recorded after activating the ReferenceClock will no longer be comparable with data acquired beforehand from the same Time Tagger instance.

3.5.3 Connecting to multiple Time Tagger Servers over the network

Once the Time Tagger servers are running at different locations, a client PC can connect to them to retrieve and process synchronized measurement data. The connection process relies on network communication, where the client searches for available Time Tagger servers and establishes a link to them.

To automatically discover servers on the local network, the function *scanTimeTaggerServers()* is used. This function sends multicast UDP messages to detect active Time Tagger servers and retrieve their IP addresses. However, its effectiveness depends on the network configuration. Since multicast messages typically remain within the same local subnet, their ability to reach servers across different subnets depends on whether the network routers forward multicast packets. If the function does not detect any servers, it is likely that multicast routing is either not enabled or unreliable in the network. In cases where *scanTimeTaggerServers()* fails to find a server, the user needs to know the IP addresses and network ports of the servers to connect to, when calling *createTimeTaggerNetwork()* on the client PC.

Warning

When a *TimeTaggerNetwork* object is created and connected to multiple servers, the time tag streams from different locations are merged into a unified data stream for measurements. This merging process relies on a sorting algorithm and requires that all incoming time tag streams have similar timestamps, meaning the Time Taggers must be properly synchronized. If the time bases are not aligned, stream merging fails.

3.5.4 Accessing individual servers

When using a *TimeTaggerNetwork* object to connect to multiple Time Tagger servers, it is possible to access and control each individual server separately. This is particularly useful when extracting detailed device-specific information that is not channel-dependent, such as checking overflow states using methods like *getOverflows()*.

The getServers() method of the TimeTaggerNetwork class returns a list of TimeTaggerServer objects, each representing one of the connected servers. These server objects act as proxies, providing access to all relevant control functions available in TimeTaggerBase and TimeTaggerHardware, as long as the servers were created with AccessMode::Control privileges. Unlike the TimeTaggerNetwork object, a TimeTaggerServer object cannot be used to perform measurements directly. Namely, it is not possible to pass the TimeTaggerServer object on to any measurement creator.

For example, to retrieve overflow information from a specific server:

```
# Retrieve the list of connected Time Tagger servers
servers = ttn.getServers()
# Access a specific server
server_A = servers[0]
```

```
# Retrieve overflow information from the server
overflows = server_A.getOverflows()
```

It is also possible to adjust hardware settings using either the *TimeTaggerServer* object or the globally mapped channels via the *TimeTaggerNetwork* object. Both approaches yield the same result:

```
# Adjust dead time via the server object
server_A.setDeadtime(1, 1000)
# Equivalent operation using TimeTaggerNetwork global channel mapping
ttn.setDeadtime(1001, 1000)
```

With the current Software, using *TimeTaggerServer* objects for configuration is primarily a convenience, as they serve as proxies for direct interaction with specific servers. However, *TimeTaggerServer* provides four specific functions that are not available in *TimeTaggerNetwork*: getAddress(), getAccessMode(), getClientChannel(), getReferenceClockState().

3.5.5 Verification of the synchronization technology using a single Time Tagger

To evaluate the precision of the synchronization technology, a single Time Tagger can be used before considering a multi-device setup. The proposed verification consists of enabling the ReferenceClock on one external reference, typically the master if there is a hierarchy, and using a second external reference as an input to the *FrequencyStability* measurement class. This allows direct analysis of the synchronization precision using built-in measurement tools.

The measurement provides stability metrics such as Allan Deviation (ADEV), Modified Allan Deviation (MDEV), and Time Deviation (TDEV), which quantify timing fluctuations over different timescales. These results characterize the synchronization performance and provide a reference before working with multiple Time Taggers.

The first external reference is connected to an input channel and used to enable the ReferenceClock. Since all analyses are performed on the same hardware, there is no need for a PPS signal for time synchronization. The second and additional external references are fed into other input channels and analyzed using the *FrequencyStability* measurement. The measurement runs over a range of averaging times to extract stability metrics.

```
import numpy as np
import time
# Define synchronization channels
ch master = 1
ch slave = 2
# Enable the ReferenceClock using the first external reference
tt.setReferenceClock(clock_channel=ch_master,
                     clock_frequency=10e6,
                     synchronization_channel=None,
                     wait_until_locked=True)
# Define measurement steps (logarithmically spaced averaging times)
steps = np.unique(np.logspace(0, 7, 100, dtype=np.int64))
# Initialize Frequency Stability measurement on the second reference
fs = TT.FrequencyStability(tt, ch_slave, steps, average=1)
# Allow the measurement to collect data
time.sleep(100) # Adjust as needed
```

```
# Retrieve frequency stability results
obj = fs.getDataObject()
tau = obj.getTau()
ADEV = obj.getADEV()
TDEV = obj.getTDEV()
MDEV = obj.getMDEV()
```

3.5.6 Measuring synchronization precision across multiple Time Taggers

Before starting experiments where Time Taggers are deployed in remote locations, it is useful to verify the synchronization precision while the units are still physically close to each other. This ensures that the synchronization setup is working correctly before the Time Taggers are separated.

The verification follows the method described in Swabian Instruments' application note on remote synchronization. A common test signal is split using a power splitter or any other method that guarantees that identical copies of the signal are fed into both Time Taggers. The signal is then connected to an input channel of each device. The *Correlation* measurement class is used to analyze the arrival times of events recorded by both Time Taggers, providing a direct measure of synchronization precision.

The measurement is performed using the *TimeTaggerNetwork* in a dual-server setup, as discussed in the previous sections.

```
import matplotlib.pyplot as plt
# Define the correlation measurement between input channels on both Time Taggers
# We assume the test signal is fed into the third input of each Time Tagger
corr = TimeTagger.Correlation(ttn, 1003, 2003, binwidth=1, n_bins=1000)
# Start the measurement and run it for a given time
corr.startFor(10e12)
corr.waitUntilFinished()
# Retrieve the correlation data
index = corr.getIndex()
counts = corr.getData()
# Plot the correlation result
plt.plot(index, counts)
plt.xlabel("Time Difference (ps)")
plt.ylabel("Counts")
plt.title("Correlation of Synchronized Time Taggers")
plt.grid()
plt.show()
```



3.6 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.

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. Network Time Tagger is an ideal solution for sharing a Time Tagger between different computers. Moreover, it is the only way to connect to multiple Server Time Taggers with a single software object, enabling the processing of Time Tag streams from different servers on-thefly. On the other hand, Pyro 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.

3.6.1 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)
```

3.6.2 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)
x = hist.getIndex()
while hist.isRunning():
    plt.pause(0.1)
```

y = hist.getData()
plt.plot(x, y)

3.6.3 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.

3.6.4 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
```

3.6.5 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.

print(uri)
Start the server event loop
daemon.requestLoop()

Client code

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

```
import Pyro5.api
# Connect to the TimeTaggerRPC object on the server
# This line is all we need to establish remote communication
TimeTagger = Pyro5.api.Proxy("PYRO:TimeTagger@localhost:23000")
# Now, we can call methods that will be executed on the server.
# Lets check what Time Taggers are available at the server
timetaggers = TimeTagger.scanTimeTagger()
print(timetaggers)
>>> ['1740000ABC', '1750000ABC']
```

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

3.6.6 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):
        """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.
        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)
```

3.6.7 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()
@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.

3.6.8 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.

3.6.9 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.

CHAPTER

FOUR

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 160, 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 90 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 90 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)*.

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

	a	•	1 0			.	-
l'able I ·	Connections	between	the Sync	hronizer.	and	l'ime '	laggers
14010 1.	connections	0000000	the byne	monie	una	1 11110	1455010

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*.

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

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

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 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.

Note

If you are synchronizing *Time Tagger Ultra* and *Time Tagger X* devices together, the internal test signal frequency of some of them may be different than expected. The reason for this is that a single testSignalDivider is applied to the synchronized group, and the internal oscillators of the *Time Tagger Ultra* and *Time Tagger X* run at different frequencies, see: *setTestSignalDivider()*. Apart from the test signal frequency change, the functionality of the Time Taggers is not affected.

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*.

CHAPTER

HARDWARE

5.1 Operating conditions

Before installing and operating the Time Tagger, users are strongly advised to follow the guidelines for proper handling. For detailed information on the operating conditions of the Time Tagger, please consult the section *Safety & Compliance*.

5.2 Input channels

The Time Tagger has 8, 18, or 20 inputs (SMA-connectors) for *Time Tagger 20, Time Tagger Ultra*, or *Time Tagger X*, respectively. 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 8, 18, or 20 for *Time Tagger 20, Time Tagger Ultra*, or *Time Tagger X*, respectively; and falling edges correspond to respective channel numbers -1 to -8, -18, or -20 for *Time Tagger 20, Time Tagger 20, Time Tagger 20, Time Tagger 20, Time Tagger X*, respectively. Thereby, you can treat rising and falling edges in a fully equivalent fashion.

5.2.1 Electrical characteristics

Property	Time Tagger 20	Time Tagger Ultra	Time Tagger X
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

5.2.2 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 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 *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.

5.2.3 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.

	ee	
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 1: Tin	ne Tagger	Ultra	Performance
--------------	-----------	-------	-------------

Table 2:	Time	Tagger	Х
----------	------	--------	---

Mode	HighRes channels	Standard channels
Standard		1 - 4, (5 - 20)
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.

5.3 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 ≥ 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 90 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.

5.4 LEDs

The Time Tagger devices have LEDs showing status information.

5.4.1 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:

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 3: Power button LED

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

Table 4: Channel LEDs

Rear panel

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 5: LED next to the CLK IN input

Table 6: LED next to the SYNC IN input

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

Table 7: LED next to the LOOP IN input

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

5.4.2 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)
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)< td=""><td>Invalid signal at SYNC IN (AUX IN 1)</td></v1.5)<>	Invalid signal at SYNC IN (AUX IN 1)
blinking yellow (hardware <v1.5)< td=""><td>Invalid signal at LOOP IN (AUX IN 2)</td></v1.5)<>	Invalid signal at LOOP IN (AUX IN 2)

Table 9: LED next to the CLK input

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

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

Table 11: LED next to the *LOOP IN* input (hardware >=v1.5)

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

5.4.3 Time Tagger 20

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

Table 12: Status LED

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

5.5 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.

5.6 Synthetic input delay

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

5.7 Synthetic dead time

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

5.8 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.

5.9 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.

5.10 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.

5.11 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 90 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 *TimeTaggerSource::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.

5.12 External Clock Input

Note

An alternative and more flexible way to apply an external clock signal is the use of *TimeTaggerSource::setReferenceClock()*. 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 *setReferenceClock()*.

Time Tagger 20

The Time Tagger 20 supports software clock feature only.

5.13 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.

5.14 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.

5.15 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.

5.16 19-inch rack mount

Time Tagger X

The *Time Tagger X* can either be installed in a 19-inch rack, requiring two height units, or operated as a tabletop instrument.

Time Tagger Ultra

Swabian Instruments offers an extra housing to make the tabletop *Time Tagger Ultra* mountable in a 19-inch rack.



Fig. 1: Rack mount for up to two Time Tagger Ultra and one Synchronizer

Parameter	Value
Channels	Up to 18 with one <i>Time Tagger Ultra</i> installed Up to 36 with two <i>Time Tagger Ultra</i> installed
Height units	3
Depth	37 cm
Weight without de- vices	3.5 kg

Table 13: Specifications of the Time Tagger Ultra rack mount

Please get in touch with *sales@swabianinstruments.com* for additional information.

Time Tagger 20

Swabian Instruments doesn't provide an option to mount the *Time Tagger 20* in a 19-inch rack.

CHAPTER

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, crosscorrelation, 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.

CHAPTER

SEVEN

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 V
tagger.setTriggerLevel(2, 0.25)
# Add time delay of 123 picoseconds on the channel 3
tagger.setInputDelay(3, 123)
# Create Correlation measurement for events in channels 2 and 3
corr = Correlation(tagger, 2, 3, binwidth=10, n_bins=1000)
# Run Correlation for 1 second to accumulate the data
corr.startFor(int(1e12), clear=True)
```

(continues on next page)

(continued from previous page)

```
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())
>> [ 800008.81 800008.81 1600017.62]
```

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

7.1.3 Using multiple Time Taggers

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

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

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

```
# Connected to the hardware as usual
tagger = createTimeTagger()
# Start the server with full remote control enabled
tagger.startServer(AccessMode.Control)
# Keep this process running
input('Press ENTER to exit the server process...')
# Stop the server if user pressed ENTER key
tagger.stopServer()
# Disconnect from the hardware
freeTimeTagger(tagger)
```

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

```
# Server address, we assume it runs on the same computer
address = 'localhost'
# Connect to the server
ttn = createTimeTaggerNetwork(address)
# Enable test signal on the remote hardware
ttn.setTestSignal(1, True)
ttn.setTestSignal(2, True)
# Create `Countrate` measurement and run it for a fixed duration
cr = Countrate(ttn, [1,2,3])
cr.startFor(1e12)
cr.waitUntilFinished()
# Print the resulting data
print(cr.getData())
# Close the connection to the server
freeTimeTagger(ttn)
```

7.2 The TimeTagger Library

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

7.2.1 Units of measurement

timestamp_t

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

channel_t

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

constexpr channel_t 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

enum 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.

Values:

enumerator Listen

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

enumerator SynchronousListen

The same as AccessMode::Listen 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.

enumerator Control

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

enumerator 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.

enum class ChannelEdge : int

Selects the channels that TimeTaggerHardware::getChannelList() returns.

Values:

enumerator All

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

enumerator Rising

Rising edges of channels with HighRes and Standard resolution.

enumerator Falling

Falling edges of channels with HighRes and Standard resolution.

enumerator HighResAll

Rising and falling of channels edges with HighRes resolution.

enumerator HighResRising

Rising edges of channels with HighRes resolution.

enumerator HighResFalling

Falling edges of channels with HighRes resolution.

enumerator StandardAll

Rising and falling edges of channels with Standard resolution.

enumerator StandardRising

Rising edges of channels with Standard resolution.

enumerator StandardFalling

Falling edges of channels with Standard resolution.

enum class CoincidenceTimestamp

Defines what timestamp to use for a coincidence event in *Coincidence / Coincidences*.

Values:

enumerator Last

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

enumerator Average

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

enumerator First

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

enumerator ListedFirst

Use the timestamp of the channel at the first position of the list when *Coincidence* or a group of *Coincidences* is instantiated.

enum class FpgaLinkInterface

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

Values:

enumerator SFPP_10GE

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

enumerator QSFPP_40GE

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

enum class GatedChannelInitial

The initial state of a GatedChannel.

Values:

enumerator Closed

The gate is closed initially.

enumerator Open

The gate is open initially.

enum 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*.

Values:

enumerator Standard

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

enumerator 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.

enumerator 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.

enumerator 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.

enum class Tag::Type

Specifies the type of a time tag.

Values:

enumerator TimeTag

Indicates a standard event corresponding to a detected signal edge.

enumerator Error

Indicates a hardware or communication-related error condition (e.g., plugging an external clock source, invalidating the global time base).

enumerator OverflowBegin

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

enumerator OverflowEnd

Marks the point where the internal overflow condition ended, resuming normal event tagging.

enumerator MissedEvents

A virtual event indicating the number of lost events per channel within an overflow interval. This event might be sent repeatedly if the number of lost events is large.

enum class UsageStatisticsStatus

Values:

enumerator **Disabled**

Usage statistics collection and upload is disabled.

enumerator Collecting

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

enumerator CollectingAndUploading

Enable usage statistics collection and automatic upload

7.2.6 Functions

TimeTagger createTimeTagger(str serial = "", Resolution 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** Serial number string of the device or empty string.
- **resolution** Select the resolution of the Time Tagger. The default is *Resolution::Standard*.

Throws

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

Returns

A Time Tagger object

```
TimeTaggerVirtual createTimeTaggerVirtual (str filename = "", timestamp_t begin = 0, timestamp_t duration = -1)
```

Creates a *TimeTaggerVirtual* object. Virtual Time Tagger uses files generated by the *FileWriter* as data source instead of the 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.

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
replay_source.appendFile('filename.ttbin')
# Replay a single file "filename.1.ttbin"
replay_source.appendFile('filename.1.ttbin')
```

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

Parameters

• **filename** – File name of the initial file. This file defines the available external channels. Default is an empty string which creates a simulated Time Tagger with 18 channels.

- begin Time offset from the beginning of the file in ps to start the replay at. Default is 0.
- **duration** Duration in picoseconds to be read from the file. *duration=-1* will replay everything. Default is -1.

Time Tagger Virtual object

TimeTaggerNetwork createTimeTaggerNetwork(str[] addresses)

Creates a new *TimeTaggerNetwork* object. During creation, the object tries to open a connection to the specified Time Tagger servers that have been created by *TimeTagger::startServer()*. This makes the remote time-tag stream locally available.

When more than one server is specified, the channel number of the n-th server is offset by n * 1000, e.g., channel 3 of the second server will become 2003. Moreover, the different servers must be synchronized to an external clock. If the connection fails, the method will throw an exception.

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

Parameters

addresses - IP addresses, or hostnames, of the servers. Use hostname:port for each server.

Throws

- RuntimeError if the connection to the server cannot be made.
- RuntimeError if the address string has an invalid format.
- RuntimeError if attempting to connect to multiple non-synchronized servers.

Returns

Time Tagger Network object

str getTimeTaggerServerInfo(str address = "localhost:41101")

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

Parameters

address – 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.

Throws

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

Returns

Information about server, available channels and exposed channels.

void freeTimeTagger(TimeTaggerBase tagger)

Releases all Time Tagger resources and terminates the active connection.

Parameters

tagger - Time Tagger Base object to disconnect

str[] scanTimeTagger()

Returns a list of the serial numbers of the connected but not instantiated Time Taggers. It may return serials blocked by other processes or already disconnected some milliseconds later.

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

List of serial numbers

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.

logger_callback setLogger(logger_callback 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)
```

void 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.

Attention

The method must be called before the first call to createTimeTagger.

int getTimeTaggerChannelNumberScheme()

Deprecated:

Since version 2.17 all values of scheme except for 1 are deprecated

Returns the currently used channel scheme.

Channel scheme

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 *TimeTaggerBase::setSoftwareClock()*.

Parameters

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

str getVersion()

Get the version of the Time Tagger software installed.

Returns

Version of the Time Tagger software.

Usage statistics data collection

See also the section Usage Statistics Collection.

void setUsageStatisticsStatus(UsageStatisticsStatus new_status)

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.

Parameters

new_status – New status of the usage statistics data collection.

UsageStatisticsStatus getUsageStatisticsStatus()

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

Current status of the usage statistics for the current user.

str 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.

7.2.7 Helper classes

class ChannelGate

Public Functions

ChannelGate(*channel_t* gate_open_channel, *channel_t* gate_close_channel, *GatedChannelInitial* initial = GatedChannelInitial::Open)

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.

Parameters

- gate_open_channel Number of the channel that opens the evaluation gate.
- gate_close_channel Number of the channel that closes the evaluation gate.
- **initial** Initial state of the evaluation gate.

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

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 *startServer()*. The *TimeTaggerNetwork* object is created with *createTimeTaggerNetwork()* which also establishes a client connection to the server.

All these objects share a common interface defined by the *TimeTaggerBase* and *TimeTaggerSource* classes. In addition, hardware-specific methods, for use with *TimeTagger* and *TimeTaggerNetwork* objects, are defined in the *TimeTaggerHardware* class.

7.3.1 General Time Tagger features

class TimeTaggerSource

This class defines methods used to configure a source of time tags, being either a *TimeTaggerBase* object, or a TimeTagger-like object such as a *TimeTaggerServer*. All Time Tagger classes implement these methods by subclassing *TimeTaggerBase* which itself subclasses *TimeTaggerSource*.

Subclassed by TimeTaggerBase, TimeTaggerServer

Public Functions

void setInputDelay(channel_t channel, timestamp_t delay)

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

Parameters

- channel Channel number.
- delay Delay time in picoseconds.

timestamp_t getInputDelay(channel_t channel)

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

Parameters

channel – Channel number.

Returns

Delay time in picoseconds.

void setDelayHardware(channel_t channel, timestamp_t delay)

Sets 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.

Note

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

Parameters

- channel Channel number.
- **delay** Delay time in picoseconds, the maximum/minimum value allowed is $\pm 2000000 (\pm 2 \ \mu s)$.

timestamp_t getDelayHardware(channel_t channel)

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

Note

Method is not available for the Time Tagger 20.

Parameters channel – Channel number.

Delay time in picoseconds.

timestamp_t[] getDelayHardwareRange(channel_t channel)

Returns a vector containing the minimum and the maximum allowable values for the hardware input delay for the specified channel.

Parameters

channel – Channel number.

Returns

Minimum and maximum hardware input delay in picoseconds.

void setDelaySoftware(channel_t channel, timestamp_t delay)

Sets 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.

Note

This method has the best performance when less than 100 events arrive within the time of the largest delay set. For example, if the rate over all channels used is 10 MTags/s, the signal can be delayed efficiently up to 10 µs. For larger delays, please consider using *DelayedChannel* instead.

Parameters

- **channel** Channel number.
- **delay** Delay time in picoseconds.

timestamp_t getDelaySoftware(channel_t channel)

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

Parameters channel – Channel number.

Returns

Delay time in picoseconds.

timestamp_t setDeadtime(channel_t channel, timestamp_t 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** Channel number.
- deadtime 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).

timestamp_t getDeadtime(channel_t channel)

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

Parameters

channel – Channel number.

Returns

Dead time value in picoseconds.

timestamp_t[] getDeadtimeRange(channel_t channel)

Returns a vector containing the minimum and the maximum allowable values for the dead time for the specified channel.

Parameters channel – Channel number.

Returns

Minimum and maximum dead time values in picoseconds.

void setConditionalFilter(channel_t[] trigger, channel_t[] filtered)

Activates or deactivates the conditional 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 of channel numbers
- filtered List of channel numbers

void clearConditionalFilter()

Deactivates the event filter. Equivalent to setConditionalFilter([], []).

channel_t[] getConditionalFilterTrigger()

Returns the collection of trigger channels for the conditional filter.

Returns

List of channel numbers.

channel_t[] getConditionalFilterFiltered()

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

Returns

List of channel numbers.

void setEventDivider(channel_t channel, int 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.

Also note that the event divider is not supported on the *TimeTaggerVirtual*.

Parameters

- **channel** Physical channel number.
- divider Divider factor, min. 1 and max. 65535.

int getEventDivider(channel_t channel)

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

See setEventDivider for further details.

Parameters channel – Physical channel number.

Returns

Divider factor value.

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.

int getOverflowsAndClear()

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

Returns

Number of overflows.

void clearOverflows()

Sets the overflow counter to zero.

void **setReferenceClock**(*channel_t* clock_channel, float clock_frequency = 10e6, float time_constant =

1e-3, *channel_t* synchronization_channel = *CHANNEL_UNUSED*, *timestamp_t* synchronization offset = 0, bool wait until locked = true)

Defines in software one of the input channels as the base clock for all channels. This feature sets up a software phase-locked loop (PLL) and rescales all incoming time-tags according to the time base provided by the *clock_channel*. This clock frequency alignment is called "syntonization".

The Reference Clock is able to handle signals decimated by *setEventDivider()* and it is possible to recover the dismissed tags in software. The new time base is characterized by "ideal clock tags" separated by exactly the defined clock_period = 1E12/clock_frequency. For measurements, you can use both, rescaled and ideal clock tags. The injection of ideal clock tags can be controlled by *setConditionalFilter()*, by default all tags are injected.

While the PLL is enabled but not locked, the time base of the instrument is invalid. In this case, the timetag stream changes to the overflow mode. This means that after a call to *setReferenceClock()*, you will typically find overflows because the PLL starts from an unlocked state.

Beyond the clock syntonization, the Reference Clock can also take a *synchronization_channel* to align the absolute time base of the Time Tagger to an external time base. Currently, the synchronization channel expects one pulse per second (1PPS) that is aligned precisely to a UTC second. The corresponding UTC second is retrieved from the computer's system clock which requires the use of a time standard via PTP. The synchronization feature is important for merging time-tag streams in the *TimeTaggerNetwork*.

Warning

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

Parameters

- **clock_channel** The physical channel that is used as reference clock input.
- **clock_frequency** The frequency of the reference clock. The value should not deviate from the real frequency by more than a few percent. If the *Event* Divider is active on this channel, you still provide the original input frequency. Default: 10E6, for 10 MHz.
- **time_constant** The time period to average over in seconds. The suppression of discretization noise is improved by a higher *time_constant*. 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: 1E-3, for 1 ms.
- **synchronization_channel** The physical channel that provides a 1PPS signal representing a UTC second.
- **synchronization_offset** Sets a manual offset to the computer's system time in ps. This is necessary if the system time is badly aligned to the 1PPS signal of the synchronization system. As this might change from start-up to start-up, it is recommended to synchronize both, the synchronization system and the computer's system time, to UTC. Default is 0.
- wait_until_locked 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

void disableReferenceClock()

Disable the software-defined reference clock.

ReferenceClockState getReferenceClockState()

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

class TimeTaggerBase : public virtual TimeTaggerSource

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

Subclassed by TimeTagger, TimeTaggerNetwork, TimeTaggerVirtual

Public Functions

Defines in software one of the input channels as the base clock for all channels. This feature sets up a software phase-locked loop (PLL) 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.

Warning

It is often useful to apply this feature in combination with *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 *setConditionalFilter()*.

Warning

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

Parameters

- input_channel The physical channel that is used as software clock input.
- **input_frequency** The frequency of the software clock after application of *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** 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 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.

void disableSoftwareClock()

Disable the software clock.

Deprecated:

use disableReferenceClock

SoftwareClockState 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.

Deprecated:

use getReferenceClock

Returns

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

int getFence(bool alloc_fence = true)

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

Parameters

alloc_fence – Optional. If False, a reference to the most recently created fence will be returned instead. Default: True.

Returns

The allocated fence.

bool waitForFence(int fence, int timeout = -1)

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

Parameters

- **fence** Fence object, which shall be waited on.
- **timeout** Optional. Timeout in milliseconds. Negative means no timeout, zero returns immediately. Default: -1.

Returns

True if the fence has passed, false on timeout.

bool sync(int timeout = -1)

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 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()*.

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* or *AccessMode::SynchronousControl*, the synchronization will be done all way through the server and the hardware. If the Time Tagger server started in *AccessMode::Listen* or *AccessMode::SynchronousListen*, 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(), TimeTagger::startServer(), AccessMode
- Synchronization of the Time Tagger pipeline

Parameters

timeout – Optional. Timeout in milliseconds. Negative means no timeout, zero returns immediately. Default: -1.

True if the synchronization was successful, false on timeout.

channel_t getInvertedChannel(channel_t 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 – Channel number.

Returns

The inverted channel number.

bool isUnusedChannel(channel_t channel)

Returns true if the passed channel number is CHANNEL_UNUSED.

Parameters

channel – Channel number.

Returns

True/False.

str 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.

xtra methods

Note

The following xtra methods are mainly for development purposes and may be discontinued in future software versions without further notice.

void xtra_setAutoStart(bool auto_start)

Configures if the new measurements and virtual channels start automatically upon creation. This is true by default for all measurements and virtual channels in all TimeTaggers, but disabled by default for the *SynchronizedMeasurements::getTagger()* proxy objects.

Warning

Disabling the auto start of new measurements and virtual channels is not recommended for most of the use cases and requires the user to start all the defined measurements and virtual channels in the correct order manually, with explicit calls to *IteratorBase::start()* or *IteratorBase::startFor()*,

Parameters

auto_start – Select whether the auto start of new measurements and virtual channel is enable.

bool xtra_getAutoStart()

Returns whether the auto start of new measurements and virtual channels is enabled.

Returns

The current auto start state of new measurements and virtual channel.

7.3.2 Time Tagger

class TimeTaggerHardware

This class provides the basic methods for configuring *TimeTagger* hardware, directly via USB or over the network.

Subclassed by TimeTagger, TimeTaggerNetwork, TimeTaggerServer

Public Functions

void setTriggerLevel(channel_t channel, float voltage)

Set the trigger level of an input channel in Volts.

Parameters

- channel Physical channel number
- **voltage** Trigger level in Volts

float getTriggerLevel(channel_t channel)

Returns trigger level for the specified physical channel number.

Parameters

channel – Physical channel number

Returns

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

timestamp_t getHardwareDelayCompensation(channel_t channel)

Get the hardware input delay compensation for the given *channel* in picoseconds. This compensation can be understood as an implicit part of *TimeTaggerSource::setDelayHardware()* and *TimeTaggerSource::setDelaySoftware()*. If your device is able to set an arbitrary delay onboard, this applies to the hardware delay compensation as well.

Parameters

channel – Physical channel number

Returns

Hardware delay compensation in picoseconds

void setHardwareDelayCompensationActive(bool use_compensation)

Activate the hardware delay compensation of a channel

The physical input delays are calibrated and compensated. However this compensation is implemented after the conditional filter and so affects its result. This function queries the effective input delay, which compensates the hardware delay.

Parameters

 $\textbf{use_compensation} - set \ compensation \ state$

void setInputImpedanceHigh(channel_t channel, bool high_impedance)

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

Note

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

Parameters

- channel Physical channel number.
- **high_impedance** True/False.

bool getInputImpedanceHigh(channel_t channel)

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

Note

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

Parameters

channel – Physical channel number.

Returns

State of high input impedance.

void setInputHysteresis(channel_t channel, int value)

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.

Note

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

Parameters

- **channel** Physical channel number.
- value Hysteresis voltage value in mV (1, 20, 70).

int getInputHysteresis(channel_t channel)

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

Note

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

Parameters

channel – Physical channel number.

Returns

Hysteresis voltage value in mV.

void setNormalization(channel_t[] channels, bool state)

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

Parameters

- channels List of physical channel numbers.
- **state** True/False.

bool getNormalization(channel_t channel)

Returns whether the Gaussian normalization of the detection jitter is enabled for the specified channel.

Parameters channel – The physical channel to query

chamer – The physical chamer to quer

Returns

True/False

str getSerial()

Returns the hardware serial number.

Returns

Serial number string.

str getModel()

Returns the model name as string.

Returns

Model name as string

str getPcbVersion()

Returns Time Tagger PCB (Printed circuit board) version.

Returns

PCB version.

float[] getDACRange()

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

Deprecated:

Since version 2.18. Please use getTriggerLevelRange() instead.

Minimum and maximum voltage, in volts.

float[] getTriggerLevelRange(channel_t channel)

Return a vector containing the minimum and the maximum voltage range for the trigger level of a given channel.

Parameters

channel – Physical channel number.

Returns

Minimum and maximum voltage for the given *channel*, in volts.

channel_t[] getChannelList(ChannelEdge type = ChannelEdge::All)

Returns a list of channels corresponding to the given type.

Parameters

type – Limits the returned channels to the specified channel edge type.

Returns

List of channel numbers.

void setHardwareBufferSize(int 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 – Buffer size, must be a positive number.

int getHardwareBufferSize()

Returns the maximum buffer size within the Time Tagger.

Returns

Maximum hardware buffer size.

timestamp_t getPsPerClock()

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

Returns

The clock period in picoseconds.

void setStreamBlockSize(int max_events, int max_latency)

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 $1-2 \ \mu s$.

Parameters

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

int getStreamBlockSizeEvents()

Returns the block size of the data stream. See setStreamBlockSize() for further details.

Returns

The maximum number of events within one block.

int getStreamBlockSizeLatency()

Returns the latency of the data stream. See setStreamBlockSize() for further details.

Returns

The maximum latency in milliseconds.

void setTestSignal(channel_t[] channel, bool enabled)

Connects or disconnects the channels with the on-chip uncorrelated signal generator.

Note

When used on a *TimeTaggerVirtual* object, this method activates a Gaussian signal generator with a 9 ps RMS jitter, on the specified *channel*. *This functionality is primarily for development use and is not intended for general application*.

Parameters

- channel List of physical channel numbers.
- **enabled** True/False

bool getTestSignal(channel_t channel)

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

Parameters

channel - Physical channel number.

Returns

True/False.

void setTestSignalDivider(int 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 – Frequency divisor factor.

int getTestSignalDivider()

Returns the value of test signal division factor.

Returns

The frequency divisor factor.

str 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.

str 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.

void disableLEDs (bool disabled)

Disables all channel LEDs and back LEDs.

Note

This feature currently lacks support for disabling the power LED on the *Time Tagger X*.

Parameters disabled – True/False.

void setLED(int 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 – LED bitmask.

void setSoundFrequency(int freq_hz)

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

Parameters

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

void setTimeTaggerNetworkStreamCompression(bool active)

Enables/disables the compression of TimeTags before they are streamed from the server to the clients.

Note

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 – Flag defining whether the compression is enabled (default: False).

int getChannelNumberScheme()

Fetch the configured numbering scheme for this TimeTagger object.

Please see setTimeTaggerChannelNumberScheme() for details.

class TimeTagger : public virtual TimeTaggerBase, public virtual TimeTaggerHardware

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.

Public Functions

void reset()

Reset the Time Tagger to the start-up state.

float[] autoCalibration()

Runs 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.

int[,] getDistributionCount()

Returns the calibration data represented in counts.

Returns

Calibration data in counts.

float[,] getDistributionPSecs()

Returns the calibration data represented in picoseconds.

Returns

Calibration data in picoseconds.

void enableFpgaLink(channel_t[] channels, str destination_mac, FpgaLinkInterface link_interface = FpgaLinkInterface::SFPP_10GE, bool exclusive = false)

Enable the FPGA link of the *Time Tagger X*.

Parameters

- channels List of channels, which shall be streamed over the FPGA link.
- **destination_mac** Destination MAC, use an empty string for the broadcast address of "FF:FF:FF:FF:FF:FF".
- **link_interface** Selects which interface shall be used, default is *FpgaLinkInterface::SFPP_10GE*.
- **exclusive** Determines if time tags should exclusively be transmitted over Ethernet, increasing Ethernet performance and avoiding USB issues, default is mixed USB and ethernet.

void disableFpgaLink()

Disable the FPGA link of the *Time Tagger X*.

void **startServer**(*AccessMode* access_mode, *channel_t*[] channels = *channel_t*[](), int 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*.

Throws

RuntimeError – If server is already running.

Parameters

- access_mode AccessMode in which the server should run. Either control or listen.
- **channels** Channels to be streamed. Used only when access_mode=AccessMode. Listen or access_mode=AccessMode.SynchronousListen.
- **port** Port at which this Time Tagger server will be listening on.

void stopServer()

Stops the Time Tagger server if currently running, otherwise does nothing.

bool isServerRunning()

Checks if the server is still running.

Returns

True is server is running and False otherwise.

void setServerAddress(str ip_address)

By default a Time Tagger in server mode will bind to IP address 0.0.0.0, exposing the server via all
IPv4 addresses of the local machine. This may be undesirable in the presence of multiple network hardware within the same machine. To prevent exposure to multiple networks, the binding IP address may be configured to refer to specific network hardware.

Parameters

ip_address – The IP address, or hostname.

str getServerAddress()

Gets the IP address, or hostname, to which the Time Tagger server shall bind.

Returns

The IP address, or hostname.

Device independent xtra methods

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*.

void xtra_setAvgRisingFalling(channel_t channel, bool 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.

They need to be manually delayed to be within a window of +-500 ps of error, else events might get lost. 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** The channel, on which the average value shall be returned.
- **enable** Select whether the averaging feature is enabled.

bool xtra_getAvgRisingFalling(channel_t channel)

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

Parameters

channel – The channel for which the averaging state is returned.

Returns

The current enable state.

void xtra_setHighPrioChannel(channel_t channel, bool 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 (*Tag::Type::TimeTag*). Timing information of low-priority channels is dismissed in overflow mode and only the number of counts is transmitted (*Tag::Type::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.

Warning

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-countrate channels, e.g. pixel triggers or similar control events.

Parameters

- **channel** The channel on which the high-priority state shall be enabled.
- enable Select whether high priority is enabled.

bool xtra_getHighPrioChannel(channel_t channel)

Get the priority state of a channel.

Parameters

channel – The channel for which the priority state is returned.

Returns

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

TTX-only xtra methods

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*.

void xtra_setAuxOut(int channel, bool enabled)

Enables/Disables the Aux Out signal for the specified Aux channel.

Parameters

- **channel** Aux channel number.
- enabled True/False.

bool xtra_getAuxOut(int channel)

Returns whether the Aux Out signal is enabled for the specified Aux channel.

Parameters

channel – Aux channel number.

Returns

State of the Aux Out signal.

void xtra_setAuxOutSignal(int channel, int divider, float duty_cycle = 0.5)

Sets the signal shape, i.e., duty cycle and frequency, of the Aux out signal for the specified Aux channel.

Parameters

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

int xtra_getAuxOutSignalDivider(int channel)

Returns the divider for the frequency of the Aux Out signal generator or the specified Aux channel.

Parameters

channel – Aux channel number.

Returns

Divider for the frequency of the Aux Out signal generator,

float xtra_getAuxOutSignalDutyCycle(int channel)

Returns the duty cycle of the Aux Out signal for the specified Aux channel.

Parameters

channel – Aux channel number.

Returns

Duty cycle of the Aux Out signal generator.

float xtra_measureTriggerLevel(channel_t channel)

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

Parameters channel – Channel number.

Returns

Applied voltage threshold of a channel

void xtra_setClockSource(int source)

Specifies the different clock sources:

- 0 internal clock
- 1 external clock 10 Mhz
- 2 external clock 500 MHz.

Parameters

source – Number of the clock source. Allowed values: 0, 1, 2.

int xtra_getClockSource()

Returns the used clock source:

- -1: auto selecting of below options
- 0: internal clock
- 1: external 10 MHz
- 2: external 500 MHz.

Returns

Number of the clock source.

void xtra_setClockAutoSelect(bool enabled)

Enables/Disables the auto clocking function.

Parameters

enabled - True/False.

bool xtra_getClockAutoSelect()

Returns whether the auto clocking function is enabled.

Returns

State of auto clocking.

void xtra_setClockOut(bool enabled)

Activates/Deactivates the 10 MHz clock output.

Parameters enabled – True/False.

7.3.3 The TimeTaggerVirtual class

class TimeTaggerVirtual : public virtual TimeTaggerBase

The *TimeTaggerVirtual* allows replaying earlier stored time-tag dump files created by the *FileWriter*. Using the virtual Time Tagger, you can repeat your experiment data analysis with different parameters or even perform different measurements.

Here is a minimal code snippet showing how to replay your data setting one measurement:

```
# Initialize the TimeTaggerVirtual by passing the name of the file to the_
...constructor.
virtual_tagger = TimeTagger.createTimeTaggerVirtual("filename.ttbin")
# Define all the virtual channels and measurements by passing the TimeTaggerVirtual_
...object
# to the tagger argument.
countrate = TimeTagger.Countrate(tagger=virtual_tagger, channels=[1,2])
# Start the replay of the data using the method run()
virtual_tagger.run()
# Wait until all time tags, or the selected chunk of it, are analyzed
virtual_tagger.waitUntilFinished()
# Retrieve the data
data = countrate.getData()
```

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.

Public Functions

int[] **run**(float speed = -1.0)

Start the replay at given speed factor. 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. If no file for replay is queued, speed < 0.0 is replaced by speed = 1.0 for simulations. This automatic speed selection is also the default value. Extreme slow replay speed between 0.0 and 0.1 is not supported.

Parameters

speed – Replay speed factor.

Returns

IDs of the queued files.

int **replay**(str file, *timestamp_t* begin = 0, *timestamp_t* duration = -1, bool queue = true)

Deprecated:

Since version 2.18. Please use createTimeTaggerVirtual and run()/appendFile() instead.

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.

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

Warning

Replaying data in small chunks is not recommended for long recordings. Each *replay()* call reads the file from the beginning up to the specified *begin*, even though only the data between *begin* and *begin* + *duration* is processed. This leads to significant overhead for chunks with higher *begin* values. Replaying the whole file at once is more efficient.

Parameters

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

Returns

ID of the queued file.

void stop()

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

int **appendFile**(str filename, *timestamp_t* begin = 0, *timestamp_t* duration = -1, bool clear = false)

Add a new file to the queue of files to be replayed. If the file includes channels that are not present in the initial file passed to createTimeTaggerVirtual, these channels will be ignored during replay and are not accessible by measurements.

Parameters

- filename The name of the file to be replayed.
- **begin** Duration in picoseconds to skip at the beginning of the file. A negative time will generate a pause in the replay.
- **duration** Duration in picoseconds to be read from the file. *duration=-1* will replay everything. (default: -1)
- **clear** If *True*, the current queue is cleared and the given file starts a new queue. (default: *False*, i.e. append to existing queue)

Returns

ID of the queued file.

bool waitForCompletion(int ID = 0, int timeout = -1)

Deprecated:

Since version 2.18. Use Please waitUntilFinished() instead.

Blocks the current thread until the replay is completed.

bool waitUntilFinished(int ID = 0, int 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.

Warning

Calling *waitUntilFinished()* on a paused timebase, such as before calling *run()*, will block the current thread indefinitely.

Parameters

- **ID** Selects which file to wait for. (default: 0)
- **timeout** Timeout in milliseconds.

Returns

True if the file is complete, false on timeout.

void setReplaySpeed(float speed)

Deprecated:

Since version 2.18. Please use *run()* instead.

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 – Replay speed factor.

float getReplaySpeed()

Returns the current speed factor. Please see also *setReplaySpeed()* for more details.

Returns

The replay speed factor

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 *startServer()*. Once the server is running, the clients can connect to it by calling *createTimeTaggerNetwork()* and specifying the server address. Starting with version 2.18, a *TimeTaggerNetwork* client can connect to multiple servers and merge their time tag streams, provided the servers are synchronized (e.g. using the White Rabbit protocol). Once the servers are running, the clients

can connect to them by calling *createTimeTaggerNetwork()* and specifying the servers addresses. A client can be any computer that can access the servers over the network or another process on the same computer. Servers and clients can run 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 **TimeTaggerNetwork** : public virtual *TimeTaggerBase*, public virtual *TimeTaggerHardware*

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.

Note

Although the TimeTaggerNetwork formally inherits from TimeTaggerBase, almost all methods of the hardware *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. When using the *TimeTaggerNetwork* with multiple servers, these forwarded method calls are always directed to the first connected server. An exception is *TimeTaggerHardware::getChannelList()*, which is handled on the client and returns all channels available across all connected servers. To extract information or interact with a specific server, use getServer() to access a *TimeTaggerServer* object. Some methods on *TimeTaggerNetwork* offer similar functionality to those on the hardware *TimeTagger*, but are implemented on the client side and can be recognized by the suffix *Client*. If the server is running in *AccessMode::Listen* or *AccessMode::SynchronousListen* 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 or AccessMode::SynchronousListen with a restricted set of channels and you call TimeTaggerHardware::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.

Public Functions

bool isConnected()

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

Returns

True/False.

void setDelayClient(channel_t channel, timestamp_t time)

Sets an artificial software delay per channel on the client side. To specify it on the server side, see setDelaySoftware() or setDelayHardware(). The latter is not available for the Time Tagger 20. 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** The channel number.
- **time** Delay time in picoseconds.

timestamp_t getDelayClient(channel_t channel)

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

Parameters channel – Channel number.

Returns

Input delay in picoseconds.

int 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 missing blocks of time tags occurred in all the hardware devices and on the network since the client connection or last call to *clearOverflowsClient()* or *getOverflowsAndClearClient()*.

Returns

The value of the client-side overflow counter.

int getOverflowsAndClearClient()

The same as *get0verflowsClient()* but also clears the client-side counter. See *get0verflowsClient()* for more information on client-side overflows.

void 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.

TimeTaggerServer getServer(str ip_address)

Parameters

ip_address – the IP address, or hostname, of the desired *TimeTagger* server.

Throws

ValueError – if *ip_address* does not match an address used in the call to createTimeTagger-Network().

Returns

a TimeTaggerServer object, for configuration and control purposes.

TimeTaggerServer[] getServers()

Returns

a list of pointers to all *TimeTaggerServer* objects making up the *TimeTaggerNetwork* client.

The *TimeTaggerServer* class contains all relevant control methods present in *TimeTaggerBase* and *TimeTaggerHardware*.

TimeTaggerServer objects allow control of a TimeTagger in server mode, provided has been created with *AccessMode::Control* privileges. However, unlike *TimeTaggerBase*, *TimeTaggerServer* cannot be used to perform measurements.

class **TimeTaggerServer** : public virtual *TimeTaggerHardware*, public virtual *TimeTaggerSource* Control and configure individual *TimeTagger* servers via a *TimeTaggerNetwork* object.

Public Functions

str getAddress()

Returns

the IP address, or hostname, of the server object.

AccessMode getAccessMode()

Returns

the AccessMode of the underlying *TimeTagger* server.

channel_t getClientChannel(channel_t server_channel)

Returns

the channel number on the TimeTaggerNetwork object corresponding to server_channel.

7.3.5 Additional classes

struct ReferenceClockState

Configuration State

timestamp_t clock_period

The rounded clock period matching the input frequency set in *TimeTaggerSource::setReferenceClock()*.

channel_t clock_channel

The input channel of the periodic clock signal set in *TimeTaggerSource::setReferenceClock()*.

channel_t synchronization_channel

The 1 pulse per second channel representing the (UTC) second *TimeTaggerSource::setReferenceClock()*.

channel_t ideal_clock_channel

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*.

float averaging_periods

The averaging periods set in *TimeTaggerBase::setReferenceClock()*.

timestamp_t synchronization_offset

The manual offset to the computer's system time in ps. This is necessary if the system time is badly aligned to the 1PPS signal of the synchronization system.

bool enabled

Indicates whether the reference clock is active or not.

int event_divider

The event divider of the *clock_channel*.

Runtime Information

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

bool is_locked

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

bool is_synchronized

Indicates whether the absolute timebase is aligned to the *synchronization_channel*.

int error_counter

Amount of locking errors since the last TimeTaggerBase::setReferenceClock() call.

timestamp_t last_ideal_clock_event

Timestamp of the last ideal clock event in picoseconds.

float period_error

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

float phase_error_estimation

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.

struct SoftwareClockState

The SoftwareClockState object contains the current configuration state:

Configuration State

timestamp_t clock_period

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

channel_t input_channel

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

channel_t ideal_clock_channel

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*.

float averaging_periods

The averaging periods set in *TimeTaggerBase::setSoftwareClock()*.

bool enabled

Indicates whether the software clock is active or not.

Runtime Information

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

bool is_locked

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

int error_counter

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

timestamp_t last_ideal_clock_event

Timestamp of the last ideal clock event in picoseconds.

float period_error

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

float phase_error_estimation

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 time window.

Coincidences

Detects coincidence clicks on multiple channel groups within a given time 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

Merges events from multiple channels into a single channel.

ConstantFractionDiscriminator

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

DelayedChannel

Creates a delayed replica of an input channel.

EventGenerator

Generates a signal pattern for each trigger event.

Frequency Multiplier

Multiplies the frequency of a periodic signal on a channel.

GatedChannel

Passes input signals only while the gate is open, defined by start and stop trigger channels.

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 *getConfiguration()*.

Returns

Configuration data of the virtual channel object.

Return type

dict

7.4.3 Coincidence

class Coincidence : public Coincidences



Detects coincidence clicks on two or more channels within a given window. Every time a coincidence is detected on the input channels (AND logic), *Coincidence* emits a tag on the virtual channel. The timestamp assigned to the coincidence on the virtual channel can be set using the parameter *timestamp*. By default, the timestamp from the last event received to complete the coincidence is used.

See all common methods

Public Functions

Coincidence(*TimeTaggerBase* tagger, *channel_t*[] channels, *timestamp_t* coincidenceWindow = 1000, *CoincidenceTimestamp* timestamp = *CoincidenceTimestamp::Last*)

Parameters

- tagger Time Tagger object instance.
- channels List of channels on which coincidence will be detected in the virtual channel.
- **coincidenceWindow** Maximum time between all events for a coincidence in picoseconds (default: 1000).
- timestamp Type of timestamp for the virtual channel (default: Last).

7.4.4 Coincidences

class Coincidences : public IteratorBase



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 compared to multiple virtual *Coincidence* channels. One object of the *Coincidence* class is limited to 64 unique channels in the list of channel groups (*coincidenceGroups*).

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_
\rightarrow of channel 1
coinc2_ch = coinc_chans[1] # triple coincidence in channels [2,3,5] with timestamp_
\rightarrow of channel 2
# or equivalent but less performant
coinc1 = Coincidence(tagger, [1,2], coincidenceWindow=10000,
c→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_
\rightarrow timestamp of channel 1
coinc2_ch = coinc2.getChannel() # triple coincidence in channels [2,3,5] with_
\rightarrow timestamp of channel 2
```

See all common methods

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).

Subclassed by Coincidence

Public Functions

Coincidences (*TimeTaggerBase* tagger, *channel_t*[][] coincidenceGroups, *timestamp_t* coincidenceWindow, *CoincidenceTimestamp* timestamp = *CoincidenceTimestamp::Last*)

Parameters

- tagger Time Tagger object instance.
- **coincidenceGroups** List of channel groups on which coincidence will be detected in the virtual channel.
- coincidenceWindow Maximum time between all events for a coincidence in picoseconds.
- timestamp Type of timestamp for the virtual channel (default: Last).

7.4.5 Combinations



class **Combinations** : public *IteratorBase*

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 class 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.

See all common methods

Note

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

Public Functions

Combinations(*TimeTaggerBase* tagger, *channel_t*[] channels, *timestamp_t* window_size)

Parameters

- tagger Time Tagger object instance.
- channels List of channels on which the combinations will be detected.
- **window_size** Maximum time between all events to make a combination, minimum time without any event detected before and after the combination window, expressed in picoseconds.

channel_t getChannel(channel_t[] input_channels)

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

Warning

The *Combinations* class enables the virtual channel corresponding to a specific combination of input channels only after an explicit call to *getChannel()*. This is essential to manage computational demands, as the number of possible combinations increases exponentially with the number of input channels.

Parameters

input_channels – List of channels forming the combination monitored by the returned virtual channel.

Returns

Virtual channel number monitoring the combination.

channel_t[] getChannels(channel_t[][] list_of_input_channel_sets)

Returns a list of virtual channel numbers corresponding to the combinations formed by the given list of sets of input channels.

Warning

This method can quickly increase the computational demands of the *Combinations* class, as all returned virtual channels are automatically enabled.

Parameters

list_of_input_channel_sets – List of sets of channels forming the combination monitored by the returned virtual channel.

Returns

List of virtual channel numbers monitoring the set of combinations.

channel_t[] getCombination(channel_t virtual_channel)

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

Parameters

virtual_channel – 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.

channel_t getSumChannel(int 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 – Length of the combinations monitored by the returned virtual channel.

Returns

Virtual channel number monitoring all combinations of *n_channel* clicks.

7.4.6 Combiner

class **Combiner** : public *IteratorBase*



Merges two or more channels into one. Every time an event is detected on any of the input channels (OR logic), *Combiner* emits a tag on the corresponding virtual channel.

See all common methods

Public Functions

Combiner(TimeTaggerBase tagger, channel_t[] channels)

Parameters

- tagger Time Tagger object instance.
- channels List of channels to be combined into a single virtual channel.

7.4.7 ConstantFractionDiscriminator

class ConstantFractionDiscriminator : public IteratorBase



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.

See all common methods

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 *TimeTaggerBase::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 *TimeTagger::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.

Public Functions

ConstantFractionDiscriminator(*TimeTaggerBase* tagger, *channel_t*[] channels, *timestamp_t* search_window)

Parameters

- tagger Time Tagger object instance.
- **channels** List of channels on which to perform the CFD. Specify rising edges only; corresponding falling edges will be registered automatically.
- search_window Maximum pulse duration, in picoseconds, for detection.

7.4.8 DelayedChannel

class DelayedChannel : public IteratorBase



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

See all common methods

Note

If you want to set a global delay for one or more input channels, *TimeTaggerBase::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.

Public Functions

DelayedChannel(*TimeTaggerBase* tagger, *channel_t* input_channel, *timestamp_t* delay)

Parameters

- tagger Time Tagger object instance.
- input_channel Channel to be delayed.
- **delay** Time by which the input is delayed, expressed in picoseconds.

void setDelay(timestamp_t 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 – Delay time in picoseconds.

7.4.9 EventGenerator

class **EventGenerator** : public *IteratorBase*



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*.

See all common methods

Public Functions

```
EventGenerator(TimeTaggerBase tagger, channel_t trigger_channel, timestamp_t[] pattern, int
trigger_divider = 1, int divider_offset = 0, channel_t stop_channel =
CHANNEL_UNUSED)
```

Parameters

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

7.4.10 FrequencyMultiplier

class FrequencyMultiplier : public IteratorBase



The *FrequencyMultiplier* inserts copies of the original input events from the *input_channel* and adds additional events to match the upscaling factor. The algorithm used assumes a constant frequency and calculates out of the last two incoming events the intermediate time stamps to match the frequency given by the *multiplier* parameter. The *FrequencyMultiplier* can be used to restore the actual frequency applied to an *input_channel* which was reduces via the *EventDivider* to lower the effective data rate.

See all common methods

Warning

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

Public Functions

FrequencyMultiplier(TimeTaggerBase tagger, channel_t input_channel, int multiplier)

Parameters

- tagger Time Tagger object instance.
- input_channel Channel on which the upscaling of the frequency is based on.
- multiplier Frequency upscaling factor.

7.4.11 GatedChannel

class GatedChannel : public IteratorBase



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*.

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 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 applies to all software versions starting from 2.10.8.

Public Functions

GatedChannel(*TimeTaggerBase* tagger, *channel_t* input_channel, *channel_t* gate_start_channel, *channel_t* gate_stop_channel, *GatedChannelInitial* initial = *GatedChannelInitial*::Closed)

Note

Note that *gate_stop_channel* == *gate_start_channel* will result in an exception.

Parameters

- tagger Time Tagger object instance.
- input_channel Channel which is gated.
- **gate_start_channel** Channel on which a signal detected will start the transmission of the *input_channel* through the gate.
- **gate_stop_channel** Channel on which a signal detected will stop the transmission of the *input_channel* through the gate.
- **initial** The initial state of the gate. If overflows occur, the gate will be reset to this state as well (default: *GatedChannelInitial::Closed*).



7.4.12 TriggerOnCountrate

class TriggerOnCountrate : public IteratorBase

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* and the *below* 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 a *time_window* after the instantiation of the object and will reflect the current state, so either *above* or *below*.

See all common methods

Public Functions

Parameters

- tagger Time Tagger object instance.
- **input_channel** Channel number of the channel whose count rate will control the trigger channels.
- **reference_countrate** The reference count rate in Hz that separates the *above* range from the *below* range.
- **hysteresis** The threshold count rate in Hz for transitioning to the *above* threshold state is countrate \>= reference_countrate + hysteresis, whereas it is countrate \<= reference_countrate hysteresis for transitioning to the *below* threshold state. The hysteresis avoids the emission of multiple trigger tags upon a single transition.
- **time_window** Rolling time window size in picoseconds. The count rate is analyzed within this time window and compared to the threshold count rate.

channel_t getChannelAbove()

Get the channel number of the *above* channel.

channel_t getChannelBelow()

Get the channel number of the *below* channel.

channel_t[] getChannels()

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

float getCurrentCountrate()

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

bool injectCurrentState()

Emits 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.

bool isAbove()

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

bool 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

Computes auto- and cross-correlations between channels.

CountBetweenMarkers

Counts events on a channel within gates defined by one or two marker channels.

Counter

Counts events on one or more channels using fixed-width bins and a circular buffer output.

Countrate

Measures the average event rate on one or more channels.

Dump

Deprecated - please use FileWriter instead. Writes raw time-tags to file in an uncompressed binary format.

FileReader

Reads time-tags from a compressed file written by the *FileWriter* measurement.

FileWriter

Writes time-tags into a file with a lossless compression. It replaces the Dump class.

Flim

Measures time histograms for fluorescence lifetime imaging.

FrequencyCounter

Measures frequency and phase evolution of periodic signals at periodic sampling intervals.

FrequencyStability

Analyzes the frequency stability of periodic signals at different time scales.

Histogram

Accumulates a histogram of time differences between two channels.

Histogram2D

Accumulates a 2D histogram of correlated time differences.

HistogramLogBins

Accumulates a histogram of time differences using logarithmic binning.

HistogramND

Accumulates an N-dimensional histogram of time differences.

IteratorBase

Provides a base class for implementing custom measurements.

PulsePerSecondMonitor

Monitors the timing and stability of 1 pulse-per-second (PPS) signals.

Sampler

Samples the digital state of channels triggered by another channel.

Scope

Detects signal edges for visualization, similar to a ultrafast logic analyzer.

StartStop

Accumulates a histogram of start-stop time differences between two channels.

SynchronizedMeasurements

Synchronizes multiple measurement instances for parallel acquisition and control.

TimeDifferences

Accumulates histograms of time differences between two channels, optionally swept using one or two trigger channels.

TimeDifferencesND

Accumulates multi-dimensional histograms of asynchronous time differences.

TimeTagStream

Provides low-level access to the raw time-tag stream for custom processing. See *Raw Time-Tag-Stream access* to get on overview about the possibilities for the raw time-tag-stream access.

7.5.2 Common methods

class IteratorBase

void clear()

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

void start()

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

void startFor(timestamp_t capture_duration, bool 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.

Parameters

- **capture_duration** Acquisition duration in picoseconds.
- **clear** Resets the accumulated data at the beginning (default: True).

void stop()

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

void abort()

Immediately aborts the measurement, discarding 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.

bool 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 if the measurement is still running.

bool waitUntilFinished(int 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 – Timeout in milliseconds. Negative value means no timeout, zero returns immediately.

Returns

True if the measurement has finished, False on timeout.

timestamp_t getCaptureDuration()

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

Returns

Capture duration in ps.

str 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.

7.5.3 Event counting

Countrate

class **Countrate** : public *IteratorBase*



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.

See all common methods

Public Functions

Countrate(TimeTaggerBase tagger, channel_t[] channels)

Parameters

- tagger Time tagger object instance.
- channels Channels for which the average count rate is measured.

float[] getData()

Returns

Average count rate in counts per second.

int[] getCountsTotal()

Returns

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

Counter

class **Counter** : public *IteratorBase*



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).

See all common methods

Public Functions

Counter(*TimeTaggerBase* tagger, *channel_t*[] channels, *timestamp_t* binwidth = 1000000000, int n_values = 1)

Parameters

- tagger Time tagger object.
- channels Channels used for counting tags.
- **binwidth** Bin width in ps (default: 1e9).
- **n_values** Number of bins (default: 1).
- int[,] getData(bool 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 – Controls how the counter array is filled (default: True).

Returns

An array of size *number of channels* by *n_values* containing the counts in each fully integrated bin.

timestamp_t[] 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.

float[,] getDataNormalized(bool 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*.

Parameters

rolling – Controls how the counter array is filled (default: True).

Returns

An array of size *number of channels* by *n_values* containing the count rate in Hz as a float in each fully integrated bin.

int[] 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.

CounterData getDataObject(bool remove = false)

Returns *CounterData* object containing a snapshot of the data accumulated in the *Counter* at the time this method is called.

Parameters

remove – Controls if the returned data shall be removed from the internal buffer (default: False).

Returns

A CounterData object providing access to a snapshot data.

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.

Public Functions

timestamp_t[] 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.

int[,] getData()

Returns

An array of size *number of channels* by *size()* containing only fully integrated bins.

float[,] getDataNormalized()

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

Returns

An array of size *number of channels* by *size()* containing the count rate in Hz as a float only for fully integrated bins.

float[,] getFrequency(timestamp_t time_scale = 1000000000000)

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*.

Returns

An array of size *number of channels* by *size()* containing the counts normalized to the specified time scale.

int[] getDataTotalCounts()

Returns the total number of events per channel since the last call to *IteratorBase::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.

timestamp_t[] getTime()

This is similar to getIndex() but it returns the absolute timestamps of the bins. For subsequent calls to Counter::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 *getData()* in ps.

int[] getOverflowMask()

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

Returns

An array of size *size()* containing overflow mask.

Public Members

int **size**

Number of returned bins.

int dropped_bins

Number of bins which have been dropped because n_values of the *Counter* measurement has been exceeded.

bool overflow

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

CountBetweenMarkers

class CountBetweenMarkers : public IteratorBase



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 *begin_channel* rather than the *be-gin_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.

See all common methods

Public Functions

CountBetweenMarkers(*TimeTaggerBase* tagger, *channel_t* click_channel, *channel_t* begin_channel, *channel_t* end_channel = *CHANNEL_UNUSED*, int n_values = 1000)

Parameters

- tagger Time tagger object.
- **click_channel** Channel on which clicks are received, gated by *begin_channel* and *end_channel*.
- **begin_channel** Channel that triggers the beginning of counting and stepping to the next value.
- **end_channel** Channel that triggers the end of counting (optional, default: *CHAN-NEL_UNUSED*)
- **n_values** Number of values stored (data buffer size) (default: 1000)

int[] getData()

Returns

Array of size *n_values* containing the acquired counter values.

timestamp_t[] getIndex()

Returns

Vector of size *n_values* containing the time in ps of each start click in respect to the very first start click.

timestamp_t[] getBinWidths()

Returns

Vector of size *n_values* containing the time differences of 'start -> (next start or stop)' for the acquired counter values.

bool ready()

Returns

True when the entire array is filled.

7.5.4 Time histograms

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

StartStop

class **StartStop** : public *IteratorBase*



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

See all common methods

Public Functions

Parameters

- tagger Time tagger object instance.
- click_channel Channel on which stop clicks are received.

- **start_channel** Channel on which start clicks are received (default: *CHAN-NEL_UNUSED*).
- **binwidth** Bin width in ps (default: 1000).

timestamp_t[,] 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.

Histogram

class Histogram : public IteratorBase



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.

See all common methods

Public Functions

Parameters

- **tagger** Time tagger object instance.
- click_channel Channel on which clicks are received.
- **start_channel** Channel on which start clicks are received (default: *CHAN-NEL_UNUSED*).

- **binwidth** Bin width in ps (default: 1000).
- **n_bins** The number of bins in the histogram (default: 1000).

int[] getData()

Returns

A one-dimensional array of size *n_bins* containing the histogram.

timestamp_t[] getIndex()

Returns

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

HistogramLogBins

class HistogramLogBins : public IteratorBase



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

$$H(\tau) = I(\tau) \cdot C_{\text{click}} \cdot C_{\text{start}}$$

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

$$g^{(2)}(\tau) = \frac{H(\tau)}{\widetilde{H}(\tau)}.$$

For $t \gg 10^{\exp_\text{stop}}$ 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 $H(\tau)$.

See all common methods

Public Functions

HistogramLogBins(TimeTaggerBase tagger, channel_t click_channel, channel_t start_channel, float exp_start, float exp_stop, int n_bins, ChannelGate click_gate = None, ChannelGate start gate = None)

Parameters

- tagger Time tagger object instance.
- click_channel Channel on which clicks are received.
- **start_channel** Channel on which start clicks are received.
- exp_start Exponent 10^exp_start in seconds where the very first bin begins.
- exp_stop Exponent 10^exp_stop in seconds where the very last bin ends.
- **n_bins** The number of bins in the histogram.
- **click_gate** Optional evaluation gate for the *click_channel*.
- **start_gate** Optional evaluation gate for the *start_channel*.

HistogramLogBinsData getDataObject()

Returns

A data object containing raw and normalization data.

timestamp_t[] getBinEdges()

Returns

A vector of size n_bins+1 containing the bin edges in picoseconds.

int[] getData()

Deprecated:

Since version 2.17.0. Please use getDataObject() and HistogramLogBinsData::getCounts() instead.

Returns

A one-dimensional array of size *n_bins* containing the histogram.

float[] getDataNormalizedCountsPerPs()

Deprecated:

Since version 2.17.0.

Returns

The counts normalized by the binwidth of each bin.

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.

class HistogramLogBinsData

Contains the histogram counts $H(\tau)$ and the corresponding normalization function $H(\tau)$.

Public Functions

float[] getG2()

Returns

A one-dimensional array of size *n_bins* containing the normalized histogram $H(\tau)/\widetilde{H}(\tau)$.

int[] getCounts()

Returns

A one-dimensional array of size *n_bins* containing the raw histogram $H(\tau)$.

float[] getG2Normalization()

Returns

A one-dimensional array of size *n_bins* containing the normalization $\widetilde{H}(\tau)$.

Histogram2D

class Histogram2D : public IteratorBase



This measurement is a 2-dimensional version of the *Histogram* measurement. The measurement accumulates two-dimensional histogram where stop signals from two separate channels define the bin coordinate. For in-
stance, 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.

See all common methods

Public Functions

Histogram2D(*TimeTaggerBase* tagger, *channel_t* start_channel, *channel_t* stop_channel_1, *channel_t* stop_channel_2, *timestamp_t* binwidth_1, *timestamp_t* binwidth_2, int n_bins_1, int n_bins_2)

Parameters

- tagger Time tagger object
- start_channel Channel on which start clicks are received
- stop_channel_1 Channel on which stop clicks for the time axis 1 are received
- stop_channel_2 Channel on which stop clicks for the time axis 2 are received
- **binwidth_1** Bin width in ps for the time axis 1
- **binwidth_2** Bin width in ps for the time axis 2
- n_bins_1 The number of bins along the time axis 1
- **n_bins_2** The number of bins along the time axis 2

int[,] getData()

Returns

A two-dimensional array of size *n_bins_1* by *n_bins_2* containing the 2D histogram.

timestamp_t[,,] 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.

timestamp_t[] getIndex_1()

Returns

A vector of size *n_bins_l* containing the bin locations in ps for the time axis 1.

timestamp_t[] getIndex_2()

Returns

A vector of size *n_bins_2* containing the bin locations in ps for the time axis 2.

HistogramND

class HistogramND : public IteratorBase

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.

See all common methods

Public Functions

HistogramND(TimeTaggerBase tagger, channel_t start_channel, channel_t[] stop_channels, timestamp_t[]
binwidths, int[] n_bins)

Parameters

- tagger Time tagger object.
- **start_channel** Channel on which start clicks are received.
- stop_channels Channel list on which stop clicks are received defining the time axes.
- binwidths Bin width in ps for the corresponding time axis.
- **n_bins** The number of bins along the corresponding time axis.

int[] 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.

timestamp_t[] getIndex(int dim = 0)

Returns

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

Correlation

class Correlation : public IteratorBase



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

See all common methods

Public Functions

Correlation(*TimeTaggerBase* tagger, *channel_t* channel_1, *channel_t* channel_2 = *CHANNEL_UNUSED*, *timestamp_t* binwidth = 1000, int n_bins = 1000)

Parameters

- **tagger** Time tagger object.
- **channel_1** Channel on which (stop) clicks are received.
- channel_2 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** Bin width in ps (default: 1000).
- **n_bins** The number of bins in the resulting histogram (default: 1000).

int[] getData()

Returns

A one-dimensional array of size *n_bins* containing the histogram.

float[] getDataNormalized()

Return the data normalized as:

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

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

Returns

Data normalized by the binwidth and the average count rate.

timestamp_t[] getIndex()

Returns

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

TimeDifferences

class **TimeDifferences** : public *IteratorBase*



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.

See all common methods

Public Functions

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.

Parameters

- tagger Time tagger object instance.
- click_channel Channel on which stop clicks are received.
- **start_channel** Channel that sets start times relative to which clicks on the click channel are measured.
- **next_channel** Channel that increments the histogram index.
- sync_channel Channel that resets the histogram index to zero.
- **binwidth** Binwidth in picoseconds.
- **n_bins** Number of bins in each histogram.
- **n_histograms** Number of histograms.

int[,] getData()

Returns

A two-dimensional array of size *n_bins* by *n_histograms* containing the histograms.

timestamp_t[] getIndex()

Returns

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

void setMaxCounts(int max_counts)

Sets the number of rollovers at which the measurement stops integrating. To integrate infinitely, set the value to 0, which is the default value.

Parameters

max_counts – Maximum number of sync/next clicks.

int 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 *sync_channel* is defined)
- 0 ... (*n_histograms* 1): Index of the currently processed histogram.

int getCounts()

Returns

The number of rollovers (histogram index resets).

bool ready()

Returns

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

TimeDifferencesND

class **TimeDifferencesND** : public *IteratorBase*



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 and reset the histogram index of the associated dimension. A subsequent event on the corresponding *next_channel* will start the accumulation again. The synchronization is done asynchronously, so an event on the *next_channel* increments the histogram index even if the accumulation is stopped. The accumulation will start 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.

See all common methods

Public Functions

Parameters

- tagger Time tagger object instance.
- click_channel Channel on which stop clicks are received.
- **start_channel** Channel that sets start times relative to which clicks on the click channel are measured.
- **next_channels** Vector of channels that increments the histogram index.
- sync_channels Vector of channels that resets the histogram index to zero.
- **n_histograms** Vector of numbers of histograms per dimension.
- **binwidth** Width of one histogram bin in ps.
- **n_bins** Number of bins in each histogram.

int[,] getData()

Returns

A two-dimensional array of size *n_bins* by *n_histograms* containing the histograms.

timestamp_t[] getIndex()

Returns

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

7.5.5 Fluorescence-lifetime imaging (FLIM)

This section describes the FLIM related measurements classes of the Time Tagger API.

FlimAbstract

class FlimAbstract : public IteratorBase

This is an interface class for FLIM measurements that defines common methods.

Subclassed by Flim, FlimBase

Public Functions

bool 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.

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 : public FlimAbstract

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.

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.

See all common methods

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.

Public Functions

Flim(TimeTaggerBase tagger, channel_t start_channel, channel_t click_channel, channel_t
pixel_begin_channel, int n_pixels, int n_bins, timestamp_t binwidth, channel_t pixel_end_channel =
 CHANNEL_UNUSED, channel_t frame_begin_channel = CHANNEL_UNUSED, int
 finish_after_outputframe = 0, int n_frame_average = 1, bool pre_initialize = true)

Parameters

- **tagger** Time tagger object instance.
- **start_channel** Channel on which start clicks are received for the time differences histogramming.
- **click_channel** Channel on which clicks are received for the time differences histogramming.
- pixel_begin_channel Start marker of a pixel (histogram).

- **n_pixels** Number of pixels (histograms) of one frame.
- **n_bins** Number of histogram bins for each pixel.
- binwidth Bin size in picoseconds.
- **pixel_end_channel** End marker of a pixel incoming clicks on the *click_channel* will be ignored afterwards (optional, default: *CHANNEL_UNUSED*).
- **frame_begin_channel** Start the frame, or reset the pixel index (optional, default: *CHANNEL_UNUSED*).
- **finish_after_outputframe** 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** Average multiple input frames into one output frame (default: 1).
- **pre_initialize** 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.

int[,] getCurrentFrame()

Returns

The histograms for all pixels of the currently active frame, 2D array with dimensions [n_bins, n_pixels].

FlimFrameInfo getCurrentFrameEx()

Returns

The currently active frame data with additional information collected at the same instance of time.

float[] 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.

int getFramesAcquired()

Returns

The number of frames that have been completed so far, since the creation or last clear of the object.

timestamp_t[] getIndex()

Returns

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

int[,] getReadyFrame(int index = -1)

Parameters

index – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional, default: -1).

Returns

The histograms for all pixels according to the frame index given. If *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]

FlimFrameInfo getReadyFrameEx(int index = -1)

Parameters

index – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional, default: -1).

Returns

The frame according to the index given. If *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.

float[] getReadyFrameIntensity(int index = -1)

Parameters

index – Index of the frame to be obtained. If -1, the last frame which has been completed is returned. (optional, default: -1).

Returns

The intensities according to the frame index given. If *index* is -1, it will return the intensity of the last frame, which has been completed. When *stop_after_outputframe* is 0, the index value must be -1. If index >= stop_after_outputframe, it will throw an error. The pixel intensity is defined by the number of counts acquired within the pixel divided by the respective integration time.

int[,] getSummedFrames(bool only_ready_frames = true, bool clear_summed = false)

Parameters

- **only_ready_frames** If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional, default: True).
- **clear_summed** If True, the summed frames memory will be cleared. (optional, default: False).

Returns

The histograms for all pixels. The counts within the histograms are integrated since the start or the last clear of the measurement.

FlimFrameInfo getSummedFramesEx(bool only_ready_frames = true, bool clear_summed = false)

Parameters

- **only_ready_frames** If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional, default: True).
- **clear_summed** If True, the summed frames memory will be cleared. (optional, default: False).

Returns

A sum of all acquired frames with additional information collected at the same instance of time.

float[] getSummedFramesIntensity(bool only_ready_frames = true, bool clear_summed = false)

Parameters

- **only_ready_frames** If true, only the finished frames are added. On false, the currently active frame is aggregated. (optional, default: True).
- **clear_summed** If True, the summed frames memory will be cleared. (optional, default: False).

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.

void initialize()

This function initialized the *Flim* object and starts execution. It does nothing if constructor parameter pre_initialize==True.

Protected Functions

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.

Parameters

- **frame_number** Current frame number.
- **data** 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** Start time for each pixel.
- pixel_end_times End time for each pixel.
- frame_begin_time Start time of the frame.
- **frame_end_time** End time of the frame.

FlimFrameInfo

class 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.

Public Functions

int 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.

bool 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.

int getPixelPosition()

Returns

A value which tells how many pixels were processed for this frame.

int[,] getHistograms()

Returns

All histograms of the frame, 2D array with dimensions [n_bins, n_pixels].

float[] getIntensities()

Returns

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

int[] getSummedCounts()

The summed counts of each histogram.

timestamp_t[] getPixelBegins()

An array of the start timestamps of each pixel.

timestamp_t[] getPixelEnds()

An array of the end timestamps of each pixel.

Public Members

int pixels

Number of pixels in the frame.

int **bins**

Number of bins of each histogram.

int frame_number

Current frame number.

int pixel_position

Current pixel position.

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 *FlimBase.frameReady* callback, which must be used to analyze the data.

class FlimBase : public FlimAbstract

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.

See all common methods

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.

Public Functions

FlimBase(TimeTaggerBase tagger, channel_t start_channel, channel_t click_channel, channel_t
pixel_begin_channel, int n_pixels, int n_bins, timestamp_t binwidth, channel_t pixel_end_channel
= CHANNEL_UNUSED, channel_t frame_begin_channel = CHANNEL_UNUSED, int

finish_after_outputframe = 0, int n_frame_average = 1, bool pre_initialize = true)

Parameters

- tagger Time tagger object instance.
- **start_channel** Channel on which start clicks are received for the time differences histogramming.
- **click_channel** Channel on which clicks are received for the time differences histogramming.
- pixel_begin_channel Start marker of a pixel (histogram).
- **n_pixels** Number of pixels (histograms) of one frame.
- **n_bins** Number of histogram bins for each pixel.
- **binwidth** Bin size in picoseconds.
- **pixel_end_channel** End marker of a pixel incoming clicks on the *click_channel* will be ignored afterwards (optional, default: *CHANNEL_UNUSED*).
- **frame_begin_channel** Start the frame, or reset the pixel index (optional, default: *CHANNEL_UNUSED*).
- **finish_after_outputframe** 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** Average multiple input frames into one output frame (default: 1).
- **pre_initialize** 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.

void initialize()

This function initialized the *Flim* object and starts execution. It does nothing if constructor parameter pre_initialize==True.

Protected Functions

virtual void frameReady(int frame_number, int[] data, timestamp_t[] pixel_begin_times, timestamp_t[]
pixel_end_times, timestamp_t frame_begin_time, timestamp_t frame_end_time)

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.

Parameters

- frame_number Current frame number.
- **data** 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 Start time for each pixel.
- **pixel_end_times** End time for each pixel.
- frame_begin_time Start time of the frame.
- **frame_end_time** End time of the frame.

7.5.6 Phase & frequency analysis

This section describes measurements that expect periodic signals, e.g., oscillator outputs.

FrequencyStability

class FrequencyStability : public IteratorBase

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). *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

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



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

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

See all common methods

Public Functions

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.

Parameters

- tagger Time tagger object.
- **channel** The input channel number.
- **steps** 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** The number of time-tags to average internally. This downsampling allows for a reduction of noise and memory requirements (default: 1000).
- **trace_len** Number of data points in the phase and frequency error traces, calculated from averaged data. The trace always contains the latest data (default: 1000).

FrequencyStabilityData getDataObject()

Returns

An object that allows access to the current metrics.

class FrequencyStabilityData

Public Functions

float[] 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.

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.

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 get TDEV():

Sample

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

Domain

Allan domain.

Returns

The overlapping MDEV.

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 HDEV.

float[] getSTDD()

Standard deviation of the periods.

Warning

The standard deviation is not recommended as a measure of frequency stability because it is nonconvergent for some types of noise commonly found in frequency sources, most noticeable the frequency drift.

Sample

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

Domain

Time domain.

Returns

The standard deviation.

float[] getADEVScaled()

Domain

Time domain.

Returns

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

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 $getMDEV()^* getTau() / \sqrt{3}$.

float[] getHDEVScaled()

Warning

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

Domain

Time domain.

Returns

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

float[] getTraceIndex()

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

Returns

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

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.

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.

float[] getTraceFrequencyAbsolute(float input_frequency = 0.0)

Provides the absolute frequency offset from a given *input_frequency* during the last *trace_len* + 1 averaged timestamps.

Parameters

input_frequency – Nominal frequency of the periodic signal (default: 0 Hz).

Returns

A trace of the last *trace_len* frequency data points in Hz.

FrequencyCounter



class **FrequencyCounter** : public *IteratorBase*

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.

See all common methods

Public Functions

Parameters

- tagger Time Tagger object instance.
- channels List of channels to analyze.
- **sampling_interval** The sampling interval in picoseconds. If the *Software-Clock* is active, it is recommended to set this value to an integer multiple of the *SoftwareClockState::clock_period*.
- **fitting_window** Time tags within this range around a sampling point are fitted for phase calculation.
- **n_values** Maximum number of sampling points to store.

FrequencyCounterData getDataObject(int event_divider = 1, bool remove = false, bool channels_last_dim = 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 Compensate for the *EventDivider* (default: 1).
- remove Control if data is removed from the internal buffer (default: True).
- channels_last_dim Determines the memory layout of the output data (default: False).
- If true, data is stored with channels as the last dimension (row-major order for channels).
- If false, data is stored with channels as the first dimension (column-major order for channels).

Returns

An object providing access to a snapshot data.

class FrequencyCounterData

Public Functions

timestamp_t[] 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.

timestamp_t[] getTime()

Array of timestamps of the sampling points.

Returns

The timestamps of the sampling points.

timestamp_t[,] getPeriodsCount()

The integer part of the phase, i.e. full periods of the oscillation.

Returns

Full cycles per channel and sampling point.

float[,] 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.

float[,] getPhase(float reference_frequency = 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.

float[,] getFrequency(timestamp_t time_scale = 100000000000)

The frequency derived from the accumulated phase difference since the last sampling interval. At index 0, there is no previous phase value to compare with, so the method returns an undefined value *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 *sampling_interval*.

Returns

A frequency value per channel and sampling point.

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.

int[,] 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.

Public Members

int **size**

Number of sampling points represented by the object.

timestamp_t overflow_samples

Number of sampling points affected by an overflow range since the start of the measurement.

bool align_to_reference

Indicates if the sampling grid has been aligned to the SoftwareClock.

timestamp_t sampling_interval

The sampling interval in picoseconds.

timestamp_t sample_offset

Index offset of the first sampling point in the object.

bool channels_last_dim

The memory layout of the output data:

- If True, the data is stored with channels as the last dimension (row-major order for channels).
- If False, the data is stored with channels as the first dimension (column-major order for channels).

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)

class PulsePerSecondMonitor : public IteratorBase

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).

See all common methods

Public Functions

PulsePerSecondMonitor(*TimeTaggerBase* tagger, *channel_t* reference_channel, *channel_t*[] signal_channels, str filename = "", *timestamp_t* period = 1E12)

Parameters

- tagger Time Tagger object instance.
- reference_channel The channel number corresponding to the PPS reference source.
- **signal_channels** A list of channel numbers with PPS signals to be compared to the reference.
- **filename** The name of the .csv file to store measurement data. By default, no data is written to file (default: "").
- **period** The assumed period of the reference source, typically one second, in picoseconds (default: 1e12).

PulsePerSecondData 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 – Controls if the returned data shall be removed from the internal buffer.

Returns

An object providing access to a snapshot data.

class PulsePerSecondData

Public Functions

int[] 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.

float[] getReferenceOffsets()

A list of offsets of each reference pulse with respective 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.

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.

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.

str[] 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.ssss).

Returns

A list of the UTC timestamp at processing time, for each reference pulse.

bool[] 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.

Public Members

int **size**

Number of reference pulses contained in the PulsePerSecondData object.

7.5.7 Time-tag-streaming

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

Time tag format

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

Size	Туре	Description
8 bit	enum Tag::Type	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

class TimeTagStream : public IteratorBase

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()*.

See all common methods

Public Functions

TimeTagStream(TimeTaggerBase tagger, int n_max_events, channel_t[] channels)

Parameters

- **tagger** Time tagger object instance.
- **n_max_events** Buffer size for storing time tags.
- **channels** List of channels to be captured.

TimeTagStreamBuffer 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.

int getCounts()

Returns

The number of stored tags since the last call to getData().

class TimeTagStreamBuffer

Public Functions

timestamp_t[] getTimestamps()

Returns an array of timestamps.

Returns

Event timestamps in picoseconds for all chosen channels.

channel_t[] getChannels()

Returns an array of channel numbers for every timestamp.

Returns

Channel number for each detected event.

int[] getOverflows()

Deprecated:

Since version 2.5. Please use getEventTypes() instead.

int[] getEventTypes()

Returns an array of event type for every timestamp. See, *Time tag format*. The method returns plain integers, but you can use *Tag::Type* to compare the values.

Returns

Event type value for each detected event.

int[] getMissedEvents()

Returns an array of missed event counts during an stream overflow situation.

Returns

Missed events value for each detected event.

Public Members

int **size**

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.

bool hasOverflows

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*.

timestamp_t tStart

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

timestamp_t tGetData

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

FileWriter

class FileWriter : public IteratorBase

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 *TimeTagger::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.

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 block
# 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.

See all common methods

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*.

Public Functions

FileWriter(TimeTaggerBase tagger, str filename, channel_t[] 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()*.

Parameters

- tagger The time tagger object.
- filename Name of the output file.
- channels List of real or virtual channels.

void split(str 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 – Filename of the new file. If empty, the old one will be used (default: empty).

void setMaxFileSize(int 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.

Parameters

max_file_size – Maximum file size in bytes (default: ~1 GByte).

int getMaxFileSize()

Returns

The maximal file size in bytes. See also setMaxFileSize().

int getTotalEvents()

Returns

The total number of events written into the file(s).

int getTotalSize()

Returns

The total number of bytes written into the file(s).

void setMarker(str marker)

Writes a comment into the file. While reading the file using the *FileReader*, the last marker can be extracted.

Parameters

marker – An arbitrary marker string to write at the current location in the file.

FileReader

class 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
```

(continues on next page)

(continued from previous page)

```
# 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.

Public Functions

FileReader(str[] filenames)

This is the class constructor. The *FileReader* automatically continues to read files that were split by the *FileWriter*.

Parameters

filenames – Filename(s) of the files to read.

TimeTagStreamBuffer getData(int 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 – Number of timetags to read from the file.

Returns

A buffer of size *n_events*.

bool 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.

str getConfiguration()

Returns

A JSON formatted string (dict in Python) that contains the Time Tagger configuration at the time of file creation.

channel_t[] getChannelList()

Returns

All channels available within the input file

str getLastMarker()

Returns

The last processed marker from the file (see also FileWriter::setMarker()).

Dump

class **Dump** : public *IteratorBase*

Writes the timetag stream into a file in a simple uncompressed binary format that store timetags as 128bit records, see *Time tag format*.

See all common methods

Warning

The files created with this class are not readable by *TimeTaggerVirtual* and *FileReader*. For storing time tag data intended for re-reading or postprocessing, use the *FileWriter* measurement class instead.

Public Functions

Dump(*TimeTaggerBase* tagger, str filename, int max_tags, *channel_t*[] channels = *channel_t*[]())

Parameters

- tagger Time Tagger object instance.
- filename Name of the output file.
- **max_tags** Stop after this number of tags has been dumped. Negative values will dump forever.
- **channels** List of channels which are dumped to the file (when empty or not passed all active channels are dumped).

Scope

class **Scope** : public *IteratorBase*



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.

See all common methods

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.

Public Functions

Scope(TimeTaggerBase tagger, channel_t[] event_channels, channel_t trigger_channel, timestamp_t
window_size = 1000000000, int n_traces = 1, int n_max_events = 1000)

Parameters

- tagger The time tagger object instance.
- event_channels List of channels.
- trigger_channel Channel number of the trigger signal.
- window_size Time window in picoseconds (default: 1 ms).
- **n_traces** Number of trigger events to be detected (default: 1).
- **n_max_events** Max number of events to be detected (default: 1000).

Event[][] 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.

bool ready()

Returns

Returns whether the acquisition is complete which means that all traces (n_traces) are acquired.

int triggered()

Returns

Returns number of trigger events have been captured so far.

timestamp_t getWindowSize()

Returns

Returns the *windows_size* parameter.

struct Event

Pair of the timestamp and the new state returned by Scope::getData.

Public Members

timestamp_t time

Timestamp in ps.

State state

Input state.

enum State

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

Values:

enumerator UNKNOWN

enumerator HIGH

enumerator 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.

See all common methods

Note

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

Public Functions

Sampler(TimeTaggerBase tagger, channel_t trigger, channel_t[] channels, int max_triggers)

Parameters

- tagger The time tagger object instance.
- trigger Channel number of the trigger signal.
- channels List of channels to be sampled.
- **max_triggers** The number of triggers and their respective sampled data, which is stored within the measurement class.

timestamp_t[,] getData()

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

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

Where the state means:

0 -- low
1 -- high
2 -- undefined (after overflow)

Returns

Sampled data

timestamp_t[,] 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) < \rightarrow< 1 | ... | any_undefined << 63],
[timestamp of second trigger, (state of channel 0) << 0 | (state of channel 1) < \rightarrow< 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.

7.5.8 Helper classes

SynchronizedMeasurements

class 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 *getTagger()* to every measurement you create. This will simultaneously disable their autostart and register for synchronization.

Public Functions

SynchronizedMeasurements(TimeTaggerBase tagger)

Parameters

tagger – The time tagger object instance.

TimeTaggerBase 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.

void start()

Calls IteratorBase::start() for every registered measurement in a synchronized way.

```
void startFor(timestamp_t capture_duration, bool clear = true)
```

Calls IteratorBase::startFor() for every registered measurement in a synchronized way.

Parameters

- capture_duration Acquisition duration in picoseconds.
- **clear** Resets the accumulated data at the beginning (default: True).

void stop()

Calls IteratorBase::stop() for every registered measurement in a synchronized way.

void clear()

Calls IteratorBase::clear() for every registered measurement in a synchronized way.

bool waitUntilFinished(int timeout = -1)

Equivalent to IteratorBase::waitUntilFinished() for synchronized measurements.
Parameters

timeout – Timeout in milliseconds. Negative value means no timeout, zero returns immediately.

Returns

True if the synchronized measurements have finished, False on timeout.

bool isRunning()

Calls *IteratorBase::isRunning()* for every registered measurement and returns true if any measurement is running.

void registerMeasurement(IteratorBase 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.

void unregisterMeasurement(IteratorBase 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.

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 timetags to be processed in this call of *process()*. This is an external reference that is shared with other measurements and might be overwritten for the next call. So if you need to store tags, create a copy.
- **begin_time** (*int*) The begin time of the data chunk.
- end_time (int) The end time of the data chunk

Override the *process()* method to include your data processing. The method will be called by the Time Tagger backend when a new chunk of time-tags is available. You are free to execute any code you like, but be aware that this is the critical part when it comes to performance. In Python, it is advisable to use numpy.array() for calculation or even pre-compiled code with Numba if an explicit iteration of the tags is necessary. Check the examples in your examples folder carefully on how to design the *process()* method.

Note

In Python, the *incoming_tags* are a structured Numpy array. You can access single tags as well as arrays of tag entries directly:

```
first_tag = incoming_tags[0]
all_timestamps = incoming_tags['time']
```

mutex

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

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

CHAPTER

IN DEPTH GUIDES

This section contains articles that provide in depth details on the Time Tagger hardware and software.

8.1 Software-Defined Reference Clock

8.1.1 Overview of synchronization concepts

Like many other instruments, the Time Tagger can be locked to an external clock. However, the Time Tagger offers two different concepts for synchronization.

Traditional hardware reference clock

If you look at the inputs of your Time Tagger, you will find one labeled CLK or CLK IN. This input allows you to apply an external frequency reference. If the hardware PLL (Phase-locked loop) of the device accepts the frequency, the Time Tagger switches to the external reference which takes over the clock role from the internal oscillator. This means that the FPGA (Field-programmable gate array) of the Time Tagger that performs the TDC (Time-to-digital conversion) is driven directly by the external clock. But there are three major drawbacks of this concept:

- The CLK IN only accepts specific frequencies with narrow bandwidth. For example, a *Time Tagger Ultra* or *Time Tagger X* will only accept 10 MHz or 500 MHz. A frequency of 9.5 MHz would already be not accepted. As a consequence, you will typically be not able to lock your Time Tagger to a laser system, although it might be desirable to use your laser as the master clock of your experiment.
- 2. For signals that are strongly correlated to the external reference, the self-calibration of the Time Tagger does not work properly anymore. The self-calibration relies on the assumption that the incoming events have a random distribution with respect to the TDC. This is always given if the internal oscillator is used but can fail for an externally defined FPGA clock.
- 3. Fixed loop filter in hardware: You cannot configure the cutoff frequency between your external clock and any built-in reference. For drifting references, e.g., high cutoff frequencies allow a faster tracking of the references. And for very stable references, low cutoff frequencies support a better white noise suppression.

There are still use cases that make use of the hardware clock (see Synchronizer).

Software-defined reference clock

The alternative and recommended concept is the software-defined reference clock. It has been introduced in software version 2.10 under the label *Software Clock* and has been extended in version 2.18 to the *Reference Clock*. In contrast to the traditional hardware synchronization, the TDC is always performed with respect to the internal oscillator of the Time Tagger. The external reference is applied to one of the standard signal inputs. This means, on the hardware level, it is handled like any other signal. On the software level, however, it is evaluated immediately by a software PLL. Based on the new time base provided by the PLL, the entire time-tag stream is rescaled. This means that the behavior of all virtual channel and measurement objects will feel to the user just like the external clock has been applied to the instrument itself.

8.1.2 Setting up the software-defined reference clock

After connecting a clock signal to one of the inputs of the Time Tagger, you can declare the respective input as the system clock by the *setReferenceClock()* method:

In this example, a 10 MHz clock has been connected to input 1. One important aspect is that the *clock_frequency* takes the actual frequency of the physical signal at the input. While the *Reference Clock* is active, an *Event Divider* cannot be set by *setEventDivider()*. This means that the *Reference Clock* is aware of the original frequency before division and is able to inject events at the place of dismissed events.

The locking behavior of the *Reference Clock* depends mostly on the *time_constant* parameter. It determines the time until the clock settles upon frequency changes. A small *time_constant* will follow the frequency quickly at the cost of the advantages of averaging. Large values will provide better average values but the locking might be lost if the frequency changes to fast. More than that, phase noise will affect your measurements more severely for strong averaging. Finding the ideal value depends on the characteristics of your reference. If the *Reference Clock* jumps out of the lock, the Time Tagger will switch to the overflow mode and timing information is lost.

The following picture demonstrates the effect of the *Reference Clock* for two signals that are on average 10 MHz. The picture exaggerates frequency fluctuations and measurement jitter for clarity. The actual *Reference Clock* would not be able to lock to such poor signals:

You can see that the input signal on channel #1 oscillates slower in the first half (sparse events) and faster in the second half (dense signals). For the signal on channel #5, it's exactly the opposite. Both signals have an additional measurement jitter that causes very close events and wider gaps at random places.

The lower panel shows the effect of declaring channel #1 as the *Reference Clock*. If you compare the signals on channels #1 and #5 from the lower panel to the upper one, you notice that the channel #1 is now quite homogeneous while the frequency change in channel #5 got even worse. The *Reference Clock* - and thereby the entire time base of the software representation - followed the frequency changes of channel #1. Note that on both channels, #1 and #5, the jitter of the original measurement is still present.

The bottom row shows the *ReferenceClockState::ideal_clock_channel*. While the *Reference Clock* calculates the new timebase, it can mathematically identify the expected timestamps of all events, transmitted and dismissed ones. These "ideal" time tags are separated by a numerically exact time difference. The *ReferenceClockState::ideal_clock_channel* is based on channel #1 but almost completely eliminates the measurement jitter. Therefore, it can be used as an improved replacement.

8.1.3 Technical limitations

Before setting up the *Reference Clock*, you should consider some limitations. We need to answer two questions, namely which signals you can use as clock signals and which accuracy you can achieve theoretically.



Input signal limitations

In comparison to the traditional hardware clock that would only accept 10 MHz or 500 MHz on a *Time Tagger Ultra*, the software-defined *Reference Clock* is extremely flexible. On a *Time Tagger Ultra* and *Time Tagger X*, it accepts any frequency from 1 kHz to the maximum frequencies of 475 MHz or 700 MHz, respectively. Tracking and evaluating periodic signals requires a sufficient internal oscillator, which the *Time Tagger 20* is lacking, unfortunately. For this reason, the *Time Tagger 20* will not achieve the performance discussed in this article. As soon as you set up the *Reference Clock* on a *Time Tagger 20*, you will face a warning that the resulting jitter will be bad. Whether this is acceptable or not, depends on your experiment, but we will not consider the *Time Tagger 20* here anymore.

Resolution limitations

The picture above shows the TDEV (Time deviation) of a 1 MHz signal on two inputs. It can be interpreted as the resolution limit for the various Time Tagger models using the *Reference Clock*. The TDEV is a metric that can be measured using the *FrequencyStability* measurement class. It is an estimator of the timing error you will find on events of the signal separated by *tau* under consideration of averaging all available data points. The measurement is performed by using a power splitter to route the 1 MHz signal to two inputs, 1 and 5. While these split signal is, of course, phase-locked for all times, the measurement jitter on both inputs is independent. In that sense, the question is: How much can the individual measurement error be suppressed by averaging?

In the experiment, channel 1 is used as the *clock_channel* in *setReferenceClock()*, while channel 5 is investigated by *FrequencyStability*. On the left, the *tau* axis starts at 1 µs, corresponding to the 1 MHz signal. The data point at 1 µs represents the shortest sample of subsequent events possible which does not allow averaging. Thus the data start on the left on the level of the specified timing-resolution of the respective model. Larger timescales allow for more and more averaging which improves the timing resolution. The TDEV settles between 100 and 200 femtoseconds for both, the *Time Tagger Ultra* and the *Time Tagger X*. This means, that for both models, there is a more or less common non-white noise for both while white noise can be eliminated at a rate of $\frac{1}{\sqrt{n}}$ by averaging over *n* events, this is not the case, e.g., for $\frac{1}{t}$ -noise.



8.1.4 Advanced features

Emulating the Conditional Filter

In most cases it is neither necessary nor desirable to transmit the full clock signal to the PC. For periodic signals like the clock, the straight-forward way to reduce the data rate would be the *EventDivider*. For a typical 10 MHz clock, you can easily apply an event divider of 100 without performance reduction.

But imagine a fluorescence lifetime experiment: A femtosecond-laser (which can be considered a clock as well) excites a sample, and now and then there will be a photon emitted a few nanoseconds later. In this case, one of the signals is aperiodic, and the traditional filter solution would be the *ConditionalFilter*: Every photon will allow exactly one laser event to pass to the PC. With that, you have pairs of events, in most cases one photon and one laser event. Their time differences can be easily evaluated by measurements like *Histogram*.

But this approach has two drawbacks in comparison to the *EventDivider* approach:

- 1. You have to sacrifice 50% of your used bandwidth for laser events instead of reducing it by a factor of 100 with the *EventDivider*.
- 2. You cannot exploit the improved temporal resolution of the *ReferenceClock* on the periodic channel because the transmitted signal is not periodic anymore.

For this scenario, the *ReferenceClock* includes a feature that emulates the *ConditionalFilter* behavior while it actually uses the *EventDivider*. The following code shows how to set up the emulator:

As you can see, the emulator is configured like the hardware *ConditionalFilter* by setConditionalFilter(). It works as a modification of the way events are injected in the *ReferenceClockState::ideal_clock_channel*. The dismissed events are recreated during the rescaling process and immediately filtered by trigger events. The effect is shown in the following sketch:

Note that the jitter of the measured laser events is widely eliminated in the ideal clock and that comparatively few events need to be transferred.

Averaging of rising and falling edges

On the *Time Tagger Ultra* and *Time Tagger X*, there is an experimental feature that typically allows to improve the noise characteristics of periodic signals: By using the *TimeTagger::xtra_setAvgRisingFalling()* method, you can configure the Time Tagger such that it transmits the average of a close-by rising and falling edge. This eliminates anti-correlated noise. Low-frequency noise (e.g. flicker $1/f^{\circ}$) on the analog input voltage shifts the trigger level transitions in the opposite direction on rising and falling edges (for similar rise vs fall times). For the white noise part (which is per definition uncorrelated), it yields $\sqrt{(2)}$. It is not yet fully clear which effects contribute to this improvement but the graph below clearly shows that this makes it possible to push the TDEV below 100 fs.

8.2 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.

8.2.1 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.

8.2.2 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 *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 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 *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:



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 *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).

8.2.3 Setup of the Conditional Filter

The setConditionalFilter() method expects two arguments, *trigger* and *filtered*, and accepts the optional boolean argument *hardwareDelayCompensation*:

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 *clearConditionalFilter()* method:

```
tagger.setConditionalFilter([], [])
# or
tagger.clearConditionalFilter()
```

8.3 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.

8.3.1 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.

8.3.2 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.

8.4 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 *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.

8.5 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.

8.5.1 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)
```

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.

8.5.2 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)
```

Similarly use our Time Tagger FPGA link reference design, but follow the instructions in the README for the QSFP+ interface.

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.

8.5.3 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 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

During software installation on Windows, you will be asked whether you want to join the Time Tagger improvement program. On Linux, usage statistics collection is disabled by default, and no explicit choice is requested during installation. You can enable, disable, or modify the usage statistics behavior at any later time using the programming interface (see below). Alternatively, on Windows, you can also change your decision by re-installing the Time Tagger software.

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.CollectingAndUploading)
```

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.

CHAPTER

TEN

SAFETY & COMPLIANCE

10.1 Safety and compliance guidelines

This section provides essential safety information, operating conditions, and compliance guidelines for the proper use and handling of all Swabian Instruments *Time Tagger* devices. Please review this section carefully before installation, operation, or maintenance of the device.

10.1.1 Symbols



10.1.2 Operation environment

The *Time Tagger* 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)			

 Table 1: Operation environment conditions

10.1.3 Electrostatic-sensitive device

The *Time Tagger* is a sensitive electronic device and must be handled with care. The input and output circuitry of the *Time Tagger* 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 the installation and operation of the *Time Tagger*.



The *Time Tagger* is sensitive to electrostatic discharge! Take appropriate protective measures when performing system installation or maintenance.

Cleaning

Before cleaning the *Time Tagger*, please make sure that it has been switched off and disconnected from all electrical connections. Clean the outer housing with a soft, lint-free dust cloth. No parts of the *Time Tagger* may be cleaned with chemical cleaning agents.

10.1.4 Disposal and recycling



Never dispose of the *Time Tagger* with the household waste. Please inform yourself about the local rules for the separate collection of electrical and electronic products.

10.1.5 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

10.2 Time Tagger X safety notice

Swabian Instruments' *Time Tagger X* is a high-resolution streaming time-to-digital converter in a 19-inch rack mountable housing. This section gives further information on the *Time Tagger X* and its operation.

10.2.1 Electrical characteristics

Parameter	Value
Voltage	100 - 240 V AC
Frequency	47-63 Hz
Power	max. 60 W

Table 2: Power supply ratings

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. The details on the characteristics of the input channels can be found in the dedicated section *Input channels*.

10.2.2 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 to 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 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.

After connecting the *Time Tagger X* to the mains, it will be in a "Standby" power mode, as 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.



Fig. 1: View of the *Time Tagger X* rear panel.





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).

10.2.3 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*.

5	e
Parameter	Value
Voltage Rating	250 V AC
Current Rating	500 mA
Dimensions	5 x 20 mm
Response Time	Slow Blow

Table	3:	Safety	fuse	ratings
rabic	<i>J</i> .	Surcey	ruse	raunge

CHAPTER

ELEVEN

REVISION HISTORY

11.1 V2.18.2 - 07.05.2025

Bug fixes

- Fixes a crash when aborting a *HistogramLogBins* measurement.
- Fixes a memory leak when starting a Time Tagger Network server in Time Tagger Lab.
- Fixes a crash when resetting a Time Tagger Lab workspace saved in a OneDrive synchronized folder.
- Fixes a channel mapping bug in Time Tagger Network when transferring low data rates in a multi-server configuration.

11.2 V2.18.0 - 23.04.2025

Highlights

- Support for Virtual Channels Coincidences, GatedChannel and DelayedChannel in Time Tagger Lab.
- Multi-server time tag stream merger with *TimeTaggerNetwork*.
- 20 channel support for the *Time Tagger X*.

Time Tagger

- Introduces ReferenceClock as an improvement over SoftwareClock to allow for synchronization with 1PPS signals and reconstruction of the original frequency.
- Adds a Raspberry Pi installer for TimeTagger software.
- Documentation: unifies the online user manual and the C++ API user manual into a single document.
- Documentation is now built into our Python and C# wrappers (via pydoc and XML).
- New tutorials for Time Tagger Lab and Remote synchronization.

Time Tagger Lab

- Introduces manually adjustable plot limits.
- Adds option to switch between Counter and FrequencyCounter as the auxiliary graph.
- Fixes unreliable behavior when using multiple monitors with different screen scaling settings.
- Visual improvements when using Synchronized devices.
- Includes several smaller UI and stability improvements.

Support and compatibility

- Removes installer for 32 bit Windows.
- Drops support for Ubuntu 18.04 and replaces CentOS 8 with AlmaLinux 8.
- Adds support Debian Bookworm (amd64 and arm64) and for AlmaLinux 9.
- Adds support for C# on Linux (not on Ubuntu 20.04 and Raspberry Pi).
- Adds support for Numpy 2.0 on all Linux distributions.

Improvements

- Adds the method setServerAddress() for binding to a single network interface.
- Adds the method *getServer()* providing a proxy object with the *TimeTaggerHardware* interface to control the Time Tagger device connected to the server.
- Adds the method *Combinations::getChannels()* to query many virtual channels at once.
- Adds the methods getTriggerLevelRange(), getDeadtimeRange() and getDelayHardwareRange() to query hardware limitations.
- Adds the enum *AccessMode::SynchronousListen* to Time Tagger Network for delivering data synchronously to clients while restricting control settings and exposing only a limited set of channels.
- Improves the performance of FileWriter and HistogramLogBins.
- Fixes a slowdown of FrequencyStability on Windows with small values for the average parameter.
- Fixes the instantiation of *FrequencyStability* in LabView.
- Fixes input registration and unregistration by measurements if some requested channel numbers are not available.
- Fixes the behavior of *Countrate* with Error events.
- · Fixes the generation of MissedEvents in Listen mode on overload conditions for TimeTaggerNetwork.

API changes

- New API for the Time Tagger Virtual: Specify the files in *createTimeTaggerVirtual()* and call *run()* after creating all measurements.
- Split *TimeTagger* in the two interfaces *TimeTaggerSource* for controlling the on-device timestamp manipulation modules and *TimeTaggerHardware* for controlling the physical parameters of a Time Tagger device.
- Support for *setTestSignal* and *getTestSignal* has been removed from *SynchronizedMeasurements* and deprecated in *TimeTaggerVirtual*.
- *freeTimeTagger()* no longer supports a Boolean return value in all programming interfaces (except C# and Labview, for backward compatibility reasons). This value was used to test whether a TimeTagger in singleton mode had been properly freed, however singleton mode has been deprecated since v2.7.2.
- C#: Throws ArgumentOutOfRangeException instead of ApplicationException on unsupported function arguments.
- Deprecates getDACRange() and setSoftwareClock().
- Renames waitForCompletion() to waitUntilFinished() to match the method name of measurements.
- Splits the former single C++ measurements header in one file per measurement.

11.3 V2.17.6 - 21.01.2025

Highlight

• The maximum default *block size* is increased to 1 million events on the *Time Tagger Ultra* and *Time Tagger X*. With the default settings, the respective maximum transfer rates exceed 90 MTags/s.

11.4 V2.17.4 - 17.07.2024

Improvement

• Python: adds support for NumPy 2.0.0 (for Python versions on Linux that support NumPy 2.0.0, rebuilding the Python wrapper may be necessary. See our *documentation* for more details).

11.5 V2.17.2 - 02.07.2024

Improvement

- Adds support for Ubuntu Long Term Support release 24.04.
- Adds *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.

11.6 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^A to 10^A.

11.7 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.

11.8 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 *Time Tagger Ultra* devices with the serials starting with 17.
- Fixes GatedChannel, FrequencyMultiplier and TriggerOnCountrate 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.

11.9 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 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 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.

11.10 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*.

11.11 V2.13.2 - 22.11.2022

Highlight

• Release of *Time Tagger Lab* - the native UI for Windows.

11.12 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.

11.13 V2.12.2 - 04.10.2022

Bug fixes

- Fixes wrong Tag::Type::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.

11.14 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 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 *getTriggerLevel()*.
- Update of the USB driver for various fixes.

11.15 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.
- *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 Tag:: Type 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.

11.16 V2.10.6 - 16.03.2022

Improvements

• Adds support for *Time Tagger Ultra* with hardware revision 1.6b.
11.17 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.

11.18 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.

11.19 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 *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.

11.20 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

- getConfiguration() and 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 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 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 getConfiguration(). Now it is returning configuration data for TimeTaggerVirtual class.
- Fixed a possible crash on Python interpreter exit while running CustomMeasurement.
- Fixed sync() signaling one block too late. The fix reduces the sync, measurement start and clear times.

11.21 V2.8.4 - 04.05.2021

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

11.22 V2.8.2 - 26.04.2021

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

11.23 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 *setSoundFrequency()*. It can be used, e.g., for optical alignment purposes (count rate -> frequency).

Changes

- Split setInputDelay() into setDelayHardware() and setDelaySoftware().
- getChannelList() filter enum renamed to ChannelEdge.
- *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.

11.24 V2.7.6 - 26.04.2021

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

11.25 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.

11.26 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 getChannelList() and waitForFence() in MATLAB.
- Fixed setDeadtime() for the TimeTaggerVirtual using 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.

11.27 V2.7.0 - 01.10.2020

Highlights

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

Fixes and improvements

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

11.28 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.

11.29 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.

11.30 V2.6.6 - 10.07.2020

Highlights

- Swabian Synchronizer support. The Synchronizer hardware can combine 8 *Time Tagger Ultra* devices 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 Coincideces have an optional parameter to select which timestamp should be inserted, the last/first completing the coincidence, the average of the event timestamps, or the first of the coincidence list.
- Fixed .net/MATLAB/LabVIEW wrappers for data with empty 2D or 3D arrays
- Provide a globally registered .NET publisher policy for C#, avoiding the 'wrong dll version' message in Labview when updating the Time Tagger software
- · setConditionalFilter throws an exception when invalid arguments are applied
- Hide the warning on fetching the TimeTaggerVirtual license without an internet connection
- DelayedChannel supports a negative delay
- Performance enhancements in StartStop

11.31 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

11.32 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

11.33 V2.6.0 - 23.12.2019

Highlights

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

Known issues

· FileWriter and FileReader have a low performance on network devices

API changes

- $\bullet \ deprecated \ TimeTagStreamBuffer.getOverflows() use \ .getEventTypes() \ instead$
- 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

11.34 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

11.35 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

11.36 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

11.37 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

11.38 V2.2.2 - 13.11.2018

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

11.39 V2.2.0 - 07.11.2018

General improvements

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

- Linux package available for Ubuntu 16.04
- Support for Python 3.7

API

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

Web application

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

11.40 V2.1.6 - 17.05.2018

fixed an error with getBinWidths from CountBetweenMarkers returning wrong values

11.41 V2.1.4 - 21.03.2018

fixed bin equilibration error appearing since V2.1.0

11.42 V2.1.2 - 14.03.2018

fixed issue installing the MATLAB toolbox

11.43 V2.1.0 - 06.03.2018

Time Tagger Ultra

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

11.44 V2.0.4 - 01.02.2018

Bug fixes

· Closing the web application server window works properly now

11.45 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.

11.46 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

11.47 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

11.48 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

11.49 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

11.50 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

11.51 V1.0.0

initial release supporting python

11.52 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.

INDEX

А

AccessMode (C++ enum), 76
AccessMode::Control (C++ enumerator), 76
AccessMode::Listen (C++ enumerator), 76
AccessMode::SynchronousControl (C++ enumerator), 77
AccessMode::SynchronousListen (C++ enumerator), 76

С

channel_t (C macro), 76 channel_t (C++ class), 76 CHANNEL_UNUSED (C++ member), 76 ChannelEdge (C++ enum), 77 ChannelEdge::All (C++ enumerator), 77 ChannelEdge::Falling (C++ enumerator), 77 ChannelEdge::HighResAll(C++ enumerator), 77 ChannelEdge::HighResFalling (*C*++ *enumerator*), 77 ChannelEdge::HighResRising(C++ enumerator),77 ChannelEdge::Rising(C++ enumerator), 77 ChannelEdge::StandardAll (C++ enumerator), 77 ChannelEdge::StandardFalling (C++ enumerator), 77 ChannelEdge::StandardRising (C++ enumerator), 77 ChannelGate (C++ class), 84 ChannelGate::ChannelGate (C++ function), 84 Coincidence (C++ class), 112 Coincidence::Coincidence (C++ function), 113 Coincidences (C++ class), 113 Coincidences::Coincidences (C++ function), 114 CoincidenceTimestamp (C++ enum), 78 CoincidenceTimestamp::Average (C++ enumerator), 78 CoincidenceTimestamp::First (C++ enumerator), 78 CoincidenceTimestamp::Last(C++ enumerator), 78 CoincidenceTimestamp::ListedFirst (C++ enumerator), 78 Combinations (C++ class), 115 Combinations::Combinations (C++ function), 116

Combinations::getChannel(C++ function), 116 Combinations::getChannels(C++ function), 116 Combinations::getCombination(C++function), 116 Combinations::getSumChannel (C++ function), 117 Combiner (C++ class), 117 Combiner::Combiner (*C*++ *function*), 117 ConstantFractionDiscriminator (C++ class), 117 ConstantFractionDiscriminator::ConstantFractionDiscriminat (C++ function), 118Correlation (C++ class), 139 Correlation::Correlation (C++ function), 139 Correlation::getData(C++ function), 139 Correlation::getDataNormalized (C++ function), 139 Correlation::getIndex (C++ function), 140 CountBetweenMarkers (C++ class), 130 CountBetweenMarkers::CountBetweenMarkers (C++ function), 131CountBetweenMarkers::getBinWidths (C++ func*tion*), 131 CountBetweenMarkers::getData(C++function), 131 CountBetweenMarkers::getIndex (C++ function), 131 CountBetweenMarkers::ready (C++ function), 132 Counter (C++ class), 127 Counter::Counter (C++ function), 128 Counter::getData(C++ function), 128 Counter::getDataNormalized (C++ function), 129 Counter::getDataObject (C++ function), 129 Counter::getDataTotalCounts (C++ function), 129 Counter::getIndex (C++ function), 128 CounterData (C++ class), 129 CounterData::dropped_bins(C++ member), 130 CounterData::getData(C++ function), 129 CounterData::getDataNormalized (C++ function), 129 CounterData::getDataTotalCounts (C++ function), 130 CounterData::getFrequency (C++ function), 129 CounterData::getIndex (C++ function), 129 CounterData::getOverflowMask(C++ function), 130 CounterData::getTime (C++ function), 130

CounterData::overflow(C++ member), 130 CounterData::size (C++ member), 130 Countrate (C++ class), 127 Countrate::Countrate(C++ function), 127 Countrate::getCountsTotal (C++ function), 127 Countrate::getData(C++ function), 127 createTimeTagger (*C*++ *function*), 80 createTimeTaggerNetwork (C++ function), 81 createTimeTaggerVirtual (C++ function), 80 CustomMeasurement (built-in class), 173

D

DelayedChannel (C++ class), 119 DelayedChannel::DelayedChannel (C++ function), 119 DelayedChannel::setDelay (C++ function), 119Dump (C++ class), 168 Dump::Dump (C++ function), 168

FileReader::getChannelList(C++ function), 167

E

Flim::getFramesAcquired (C++ function), 146 Flim::getIndex (C++ function), 146 Event (C++ *struct*), 169 Flim::getReadyFrame (C++ function), 146 Event::state (C++ member), 170 Flim::getReadyFrameEx (C++ function), 146 Event::time (C++ member), 170 Flim::getReadyFrameIntensity(C++ function), 147 EventGenerator (C++ class), 119 Flim::getSummedFrames(C++ function), 147 EventGenerator::EventGenerator (C++ function), Flim::getSummedFramesEx (C++ function), 147 120 Flim::getSummedFramesIntensity (C++ function), Experimental::PulsePerSecondData (C++ class), 147 161 Flim::initialize (C++ function), 148 Experimental::PulsePerSecondData::getIndices FlimAbstract (C++ class), 143 (C++ function), 162Experimental::PulsePerSecondData::getReferenceOffSetStract::isAcquiring(C++ function), 143 FlimBase (C++ class), 149 (C++ function), 162Experimental::PulsePerSecondData::getSignalOffSetBase::FlimBase(C++ function), 150 FlimBase::frameReady (C++ function), 151 (C++ function), 162FlimBase::initialize (C++ function), 150 Experimental::PulsePerSecondData::getStatus FlimFrameInfo (C++ class), 148 (C++ function), 162Experimental::PulsePerSecondData::getUtcDates FlimFrameInfo::bins(C++ member), 149 FlimFrameInfo::frame_number (C++ member), 149 (C++ function), 162 $\label{eq:spectrum} \texttt{Experimental::PulsePerSecondData::getUtcSecondSlimFrameInfo::getFrameNumber $(C++$ function$), the second of the second secon$ 148 (C++ function), 162FlimFrameInfo::getHistograms(C++ function), 149 Experimental::PulsePerSecondData::size (C++ FlimFrameInfo::getIntensities (C++ function), *member*), 162 149 Experimental::PulsePerSecondMonitor (C++FlimFrameInfo::getPixelBegins (C++ function), *class*), 161 149 Experimental::PulsePerSecondMonitor::getDataObject FlimFrameInfo::getPixelEnds(C++function), 149 (C++ function), 161 $\label{eq:experimental::PulsePerSecondMonitor::PulsePerSecOndMonitor::PulsePerSecOndMonit$ 149 (C++ function), 161FlimFrameInfo::getSummedCounts (C++ function), External delay, 183 149 F FlimFrameInfo::isValid(C++ function), 148 FlimFrameInfo::pixel_position (C++ member), FileReader (C++ class), 166 149 FileReader::FileReader (C++ function), 167 FlimFrameInfo::pixels(C++ member), 149

FileReader::getConfiguration(C++ function), 167

FileReader::getLastMarker(C++ function), 167

FileWriter::getMaxFileSize(C++ function), 166

FileWriter::getTotalEvents (C++ function), 166

FileWriter::getTotalSize(C++ function), 166

FileWriter::setMarker (C++ function), 166 FileWriter::setMaxFileSize(C++ function), 166

Flim::getCurrentFrame (C++ function), 146

Flim::getCurrentFrameEx (C++ function), 146

Flim::getCurrentFrameIntensity (C++ function),

FileWriter::split(C++ function), 165

Flim::frameReady (C++ function), 148

Flim::Flim(*C*++*function*), 145

FileReader::getData(C++ function), 167

FileReader::hasData(C++ function), 167

FileWriter::FileWriter (C++ function), 165

FileWriter (C++ class), 164

Flim(C++ class), 145

146

FpgaLinkInterface (C++ enum), 78 FpgaLinkInterface::QSFPP_40GE (C++ enumerator), 78 FpgaLinkInterface::SFPP_10GE (C++ enumerator), 78 freeTimeTagger (C++ function), 81 FrequencyCounter (C++ class), 157 FrequencyCounter::FrequencyCounter (C++ func*tion*). 158 FrequencyCounter::getDataObject(C++ function), 158 FrequencyCounterData (C++ class), 158 FrequencyCounterData::align_to_reference (C++ member), 160FrequencyCounterData::channels_last_dim (C++ member), 160FrequencyCounterData::getFrequency (C++ function), 159 FrequencyCounterData::getFrequencyInstantaneous (C++ function), 159FrequencyCounterData::getIndex (C++ function), 158 FrequencyCounterData::getOverflowMask (C++ function), 159 FrequencyCounterData::getPeriodsCount (C++ function), 158 FrequencyCounterData::getPeriodsFraction (C++ function), 159FrequencyCounterData::getPhase (C++ function), 159 FrequencyCounterData::getTime (C++ function), 158 FrequencyCounterData::overflow_samples (C++ *member*), 160 FrequencyCounterData::sample_offset (C++*member*), 160 FrequencyCounterData::sampling_interval (C++ member), 160FrequencyCounterData::size(C++ member), 160 FrequencyMultiplier (C++ class), 120 FrequencyMultiplier::FrequencyMultiplier (C++ function), 121FrequencyStability (C++ class), 151 FrequencyStability::FrequencyStability (C++ function), 153 FrequencyStability::getDataObject (C++ function), 153 FrequencyStabilityData (C++ class), 153 FrequencyStabilityData::getADEV (C++ function), 154 FrequencyStabilityData::getADEVScaled (C++ function), 155 FrequencyStabilityData::getHDEV (C++ function), 154

FrequencyStabilityData::getHDEVScaled (C++ function), 156 FrequencyStabilityData::getMDEV (C++ function), 154 FrequencyStabilityData::getSTDD (C++ function),

155 FrequencyStabilityData::getTau (C++ function), 154

FrequencyStabilityData::getTDEV (C++ function), 155

FrequencyStabilityData::getTraceFrequency (C++ function), 156

FrequencyStabilityData::getTraceFrequencyAbsolute (C++ function), 156

FrequencyStabilityData::getTraceIndex (C++ function), 156

FrequencyStabilityData::getTracePhase (C++ function), 156

G

GatedChannel (C++ class), 121 GatedChannel::GatedChannel(C++ function), 122 GatedChannelInitial (C++ enum), 78 GatedChannelInitial::Closed (C++ enumerator), 78 GatedChannelInitial::Open(C++ enumerator), 78 getChannel() (VirtualChannel method), 112 getChannels() (VirtualChannel method), 112 getTimeTaggerChannelNumberScheme (C++ function), 82 getTimeTaggerServerInfo (C++ function), 81 getUsageStatisticsReport (C++ function), 84 getUsageStatisticsStatus (C++ function), 83 getVersion (C++ function), 83

Н

Hardware delay, 183 Histogram (C++ class), 133 Histogram2D (C++ class), 136 Histogram2D::getData(C++ function), 137 Histogram2D::getIndex (C++ function), 137 Histogram2D::getIndex_1(C++ function), 137 Histogram2D::getIndex_2 (C++ function), 137 Histogram2D::Histogram2D(C++ function), 137 Histogram::getData(C++ function), 134 Histogram::getIndex(C++ function), 134 Histogram::Histogram (C++ function), 133 HistogramLogBins (C++ class), 134 HistogramLogBins::getBinEdges (C++ function), 135 HistogramLogBins::getData(C++ function), 135 HistogramLogBins::getDataNormalizedCountsPerPs (C++ function), 135

HistogramLogBins::getDataNormalizedG2 (C++ ReferenceClockState::ideal_clock_channel function), 136 HistogramLogBins::getDataObject(C++ function), 135 HistogramLogBins::HistogramLogBins (C++ func*tion*), 135 HistogramLogBinsData (C++ class), 136 HistogramLogBinsData::getCounts (C++ function), 136 HistogramLogBinsData::getG2 (C++ function), 136 HistogramLogBinsData::getG2Normalization (C++ function), 136HistogramND (C++ class), 137 HistogramND::getData(C++ function), 138 HistogramND::getIndex(C++ function), 138 HistogramND::HistogramND (C++ function), 138

L

Input time stamp, 183 IteratorBase (built-in class), 125 IteratorBase::abort (C++ function), 126 IteratorBase::clear (C++ function), 125 IteratorBase::getCaptureDuration (C++ function), 126 IteratorBase::getConfiguration (C++ function), 126 IteratorBase::isRunning(C++ function), 126 IteratorBase::start (C++ function), 125 IteratorBase::startFor (C++ function), 125 IteratorBase::stop (C++ function), 125 IteratorBase::waitUntilFinished(C++ function), 126

Μ

mergeStreamFiles (C++ function), 83 mutex (CustomMeasurement attribute), 174

Ρ

process() (CustomMeasurement method), 174

R

ReferenceClockState (C++ struct), 109 ReferenceClockState::averaging_periods (C++ *member*), 109 ReferenceClockState::clock_channel(C++ member), 109 ReferenceClockState::clock_period (C++ member), 109 ReferenceClockState::enabled(C++ member), 109 ReferenceClockState::error_counter(C++ member). 110 ReferenceClockState::event_divider (C++ member), 109

- (C + + member), 109
- ReferenceClockState::is_locked (C++ member), 110
- ReferenceClockState::is_synchronized (C++*member*), 110
- ReferenceClockState::last_ideal_clock_event (C++ member), 110
- ReferenceClockState::period_error (C++ member), 110
- ReferenceClockState::phase_error_estimation (C + + member), 110

ReferenceClockState::synchronization_channel (C + + member), 109

- ReferenceClockState::synchronization_offset (C++ member), 109
- Resolution (C++ enum), 78
- Resolution::HighResA (C++ enumerator), 79
- Resolution::HighResB (C++ enumerator), 79 Resolution::HighResC (C++ enumerator), 79
- Resolution::Standard (C++ enumerator), 78

S

Sampler (C++ class), 170 Sampler::getData(C++ function), 171 Sampler::getDataAsMask (C++ function), 171 Sampler::Sampler (C++ function), 171 scanTimeTagger (C++ function), 81 scanTimeTaggerServers (C++ function), 82 Scope (C++ class), 168 Scope::getData (C++ function), 169 Scope::getWindowSize(C++ function), 169 Scope::ready (C++ function), 169 Scope::Scope (C++ function), 169 Scope::triggered (C++ function), 169 setLogger (C++ function), 82 setTimeTaggerChannelNumberScheme (C++ func*tion*), 82 setUsageStatisticsStatus (C++ function), 83 SoftwareClockState (C++ struct), 110 SoftwareClockState::averaging_periods (C++ *member*), 110 SoftwareClockState::clock_period (C++ member), 110 SoftwareClockState::enabled(C++ member), 110 SoftwareClockState::error_counter (C++ member), 111 SoftwareClockState::ideal_clock_channel (C + + member), 110SoftwareClockState::input_channel (C++ member), 110 SoftwareClockState::is_locked (C++ member), 111

SoftwareClockState::last_ideal_clock_event (C++ member), 111SoftwareClockState::period_error (C++ member), 111 SoftwareClockState::phase_error_estimation (C + + member), 111StartStop (C++ class), 132 StartStop::getData(C++ function), 133 StartStop::StartStop(C++ function), 132 State (C++ enum), 170 State::HIGH (C++ enumerator), 170 State::LOW (C++ enumerator), 170 State::UNKNOWN (C++ enumerator), 170 SynchronizedMeasurements (C++ class), 172 SynchronizedMeasurements::clear(C++ function), 172 SynchronizedMeasurements::getTagger (C++function), 172 SynchronizedMeasurements::isRunning (C++function), 173 SynchronizedMeasurements::registerMeasurement TimeTagger::stopServer (C++ function), 100 (C++ function), 173SynchronizedMeasurements::start (C++ function), 172 SynchronizedMeasurements::startFor (C++ function), 172 SynchronizedMeasurements::stop (C++ function), 172 SynchronizedMeasurements::SynchronizedMeasuremEintesTagger::xtra_getClockAutoSelect (C++ function), 172SynchronizedMeasurements::unregisterMeasuremenTimeTagger::xtra_getClockSource(C++ function), (C++ function), 173SynchronizedMeasurements::waitUntilFinished (C++ function), 172Т Tag::Type (C++ enum), 79

Tag::Type::Error (C++ enumerator), 79 Tag::Type::MissedEvents(C++ enumerator), 79 Tag::Type::OverflowBegin (C++ enumerator), 79 Tag::Type::OverflowEnd(C++ enumerator), 79 Tag::Type::TimeTag (C++ enumerator), 79 TDC time stamp, 183 TimeDifferences (C++ class), 140 TimeDifferences::getCounts(C++ function), 142 TimeDifferences::getData(C++ function), 141 TimeDifferences::getHistogramIndex (C++ func*tion*), 141 TimeDifferences::getIndex (C++ function), 141 TimeDifferences::ready (C++ function), 142 TimeDifferences::setMaxCounts (C++ function), 141 TimeDifferences::TimeDifferences (C++ func*tion*), 141

TimeDifferencesND (C++ class), 142 TimeDifferencesND::getData(C++ function), 143 TimeDifferencesND::getIndex (C++ function), 143 TimeDifferencesND::TimeDifferencesND (C++function), 143 timestamp_t (C macro), 75 $timestamp_t(C++ class), 75$ TimeTagger (C++ class), 99 TimeTagger::autoCalibration (C++ function), 99 TimeTagger::disableFpgaLink (C++ function), 100 TimeTagger::enableFpgaLink (C++ function), 100 TimeTagger::getDistributionCount (C++ func*tion*), 99 TimeTagger::getDistributionPSecs (C++ func*tion*), 100 TimeTagger::getServerAddress(C++ function), 101 TimeTagger::isServerRunning (C++ function), 100 TimeTagger::reset (C++ function), 99 TimeTagger::setServerAddress(C++function), 100 TimeTagger::startServer (C++ function), 100 TimeTagger::xtra_getAuxOut (C++ function), 102 TimeTagger::xtra_getAuxOutSignalDivider (C++ function), 103TimeTagger::xtra_getAuxOutSignalDutyCycle (C++ function), 103TimeTagger::xtra_getAvgRisingFalling (C++function), 101 (C++)function), 103 103 TimeTagger::xtra_getHighPrioChannel (C++function), 102 TimeTagger::xtra_measureTriggerLevel (C++function), 103 TimeTagger::xtra_setAuxOut (C++ function), 102 TimeTagger::xtra_setAuxOutSignal (C++ function), 102 TimeTagger::xtra_setAvgRisingFalling (C++function), 101 TimeTagger::xtra_setClockAutoSelect (C++function), 103 TimeTagger::xtra_setClockOut(C++function), 104 TimeTagger::xtra_setClockSource(C++ function), 103 TimeTagger::xtra_setHighPrioChannel (C++function), 101 TimeTaggerBase (built-in class), 89 TimeTaggerBase (C++ class), 89 TimeTaggerBase::disableSoftwareClock (C++function), 90 TimeTaggerBase::getConfiguration (C++ function, 92

TimeTaggerBase::getFence(*C*++ *function*), 91 TimeTaggerBase::getInvertedChannel (C++ func*tion*). 92 TimeTaggerBase::getSoftwareClockState (C++ function), 90 TimeTaggerBase::isUnusedChannel (C++ function), 92 TimeTaggerBase::setSoftwareClock (C++ func*tion*). 90 TimeTaggerBase::sync (C++ function), 91 TimeTaggerBase::waitForFence (C++ function), 91 TimeTaggerBase::xtra_getAutoStart (C++ func*tion*), 92 TimeTaggerBase::xtra_setAutoStart (C++ function), 92 TimeTaggerHardware (C++ class), 93 TimeTaggerHardware::disableLEDs (C++ function), 98 TimeTaggerHardware::getChannelList (C++ func*tion*), 96 TimeTaggerHardware::getChannelNumberScheme (C++ function), 99TimeTaggerHardware::getDACRange (C++ function), 95 TimeTaggerHardware::getDeviceLicense (C++function), 98 TimeTaggerHardware::getHardwareBufferSize (C++ function), 96TimeTaggerHardware::getHardwareDelayCompensatiDimeTaggerNetwork::getDelayClient (C++ func-(C++ function), 93TimeTaggerHardware::getInputHysteresis (C++ function), 95 TimeTaggerHardware::getInputImpedanceHigh (C++ function), 94TimeTaggerHardware::getModel (C++ function), 95 TimeTaggerHardware::getNormalization (C++function), 95 TimeTaggerHardware::getPcbVersion (C++ function), 95 TimeTaggerHardware::getPsPerClock (C++ func*tion*), 96 TimeTaggerHardware::getSensorData (C++ func*tion*). 98 TimeTaggerHardware::getSerial(C++function),95 TimeTaggerHardware::getStreamBlockSizeEvents (C++ function), 97TimeTaggerHardware::getStreamBlockSizeLatency (C++ function), 97TimeTaggerHardware::getTestSignal (C++ function), 97 TimeTaggerHardware::getTestSignalDivider (C++ function), 98TimeTaggerHardware::getTriggerLevel function), 93

TimeTaggerHardware::getTriggerLevelRange (C++ function), 96TimeTaggerHardware::setHardwareBufferSize (C++ function), 96TimeTaggerHardware::setHardwareDelayCompensationActive (C++ function), 93TimeTaggerHardware::setInputHysteresis (C++ function), 94 TimeTaggerHardware::setInputImpedanceHigh (C++ function), 93TimeTaggerHardware::setLED (C++ function), 98 TimeTaggerHardware::setNormalization (C++function), 95 TimeTaggerHardware::setSoundFrequency (C++function), 99 TimeTaggerHardware::setStreamBlockSize (C++ function), 96 TimeTaggerHardware::setTestSignal (C++ func*tion*), 97 TimeTaggerHardware::setTestSignalDivider (C++ function), 97TimeTaggerHardware::setTimeTaggerNetworkStreamCompression (C++ function), 99TimeTaggerHardware::setTriggerLevel (C++function), 93 TimeTaggerNetwork (C++ class), 107 TimeTaggerNetwork::clearOverflowsClient (C++ function), 108tion), 108 TimeTaggerNetwork::getOverflowsAndClearClient (C++ function), 108TimeTaggerNetwork::getOverflowsClient (C++ function), 108 TimeTaggerNetwork::getServer (C++ function), 108 TimeTaggerNetwork::getServers (C++ function), 108 TimeTaggerNetwork::isConnected (C++ function), 107 TimeTaggerNetwork::setDelayClient (C++ func*tion*), 107 TimeTaggerServer (C++ class), 108 TimeTaggerServer::getAccessMode (C++ function), 109 TimeTaggerServer::getAddress(C++function), 109 TimeTaggerServer::getClientChannel (C++ function), 109 TimeTaggerSource (*C*++ *class*), 85 TimeTaggerSource::clearConditionalFilter (C++ function), 87TimeTaggerSource::clearOverflows (C++ func*tion*), 88 (C++ TimeTaggerSource::disableReferenceClock

TimeTaggerSource::getConditionalFilterFilteredTimeTagStreamBuffer (C++ class), 163 (C++ function), 87TimeTagStreamBuffer::getChannels (C++ func-TimeTaggerSource::getConditionalFilterTrigger tion), 164 TimeTagStreamBuffer::getEventTypes (C++ func-(C++ function), 87TimeTaggerSource::getDeadtime(C++function),87 *tion*), 164 TimeTaggerSource::getDeadtimeRange (C++ func-TimeTagStreamBuffer::getMissedEvents (C++function), 164 *tion*). 87 TimeTaggerSource::getDelayHardware (C++ func-TimeTagStreamBuffer::getOverflows (C++ function), 85 *tion*). 164 TimeTaggerSource::getDelayHardwareRange TimeTagStreamBuffer::getTimestamps (C++ func-(C++ function), 86tion), 164 TimeTaggerSource::getDelaySoftware (C++ func-TimeTagStreamBuffer::hasOverflows (C++ mem*tion*), 86 ber), 164 TimeTaggerSource::getEventDivider (C++ func-TimeTagStreamBuffer::size(C++ member), 164 TimeTagStreamBuffer::tGetData (C++ member), tion), 88 TimeTaggerSource::getInputDelay (C++ function), 164 TimeTagStreamBuffer::tStart (C++ member), 164 85 TimeTaggerSource::getOverflows (C++ function), TriggerOnCountrate (C++ class), 122 TriggerOnCountrate::getChannelAbove 88 (C++TimeTaggerSource::getOverflowsAndClear (C++ function), 123 function), 88 TriggerOnCountrate::getChannelBelow (C++TimeTaggerSource::getReferenceClockState function), 123 (C++ function), 89TriggerOnCountrate::getChannels(C++ function), TimeTaggerSource::setConditionalFilter (C++ 123 TriggerOnCountrate::getCurrentCountrate function), 87 TimeTaggerSource::setDeadtime(C++function),86 (C++ function), 123TimeTaggerSource::setDelayHardware (C++ func-TriggerOnCountrate::injectCurrentState (C++ *tion*), 85 function), 123 TriggerOnCountrate::isAbove (C++ function), 124 TimeTaggerSource::setDelaySoftware (C++ func-TriggerOnCountrate::isBelow (C++ function), 124 tion), 86 TimeTaggerSource::setEventDivider (C++ func-TriggerOnCountrate::TriggerOnCountrate (C++ tion), 87 function), 123 TimeTaggerSource::setInputDelay (C++ function), U 85 TimeTaggerSource::setReferenceClock (C++UsageStatisticsStatus (C++ enum), 79 function), 88 UsageStatisticsStatus::Collecting (C++ enu-TimeTaggerVirtual (C++ class), 104 merator), 79 TimeTaggerVirtual::appendFile (C++ function), UsageStatisticsStatus::CollectingAndUploading 105 (C++ enumerator), 80TimeTaggerVirtual::getReplaySpeed (C++ func-UsageStatisticsStatus::Disabled (C++ enumera*tion*), 106 tor), 79 TimeTaggerVirtual::replay (C++ function), 105 TimeTaggerVirtual::run (C++ function), 104 TimeTaggerVirtual::setReplaySpeed (C++ function), 106 TimeTaggerVirtual::stop (C++ function), 105 TimeTaggerVirtual::waitForCompletion (C++function), 106 TimeTaggerVirtual::waitUntilFinished (C++function), 106 TimeTagStream (C++ class), 163 TimeTagStream::getCounts (C++ function), 163 TimeTagStream::getData(C++ function), 163 TimeTagStream::TimeTagStream(C++function), 163