

TimeTagger

2.17.4.0

Generated by Doxygen 1.10.0



<b>1 TimeTagger</b>	<b>1</b>
<b>2 Topic Index</b>	<b>3</b>
2.1 Topics . . . . .	3
<b>3 Namespace Index</b>	<b>5</b>
3.1 Namespace List . . . . .	5
<b>4 Hierarchical Index</b>	<b>7</b>
4.1 Class Hierarchy . . . . .	7
<b>5 Class Index</b>	<b>9</b>
5.1 Class List . . . . .	9
<b>6 File Index</b>	<b>13</b>
6.1 File List . . . . .	13
<b>7 Topic Documentation</b>	<b>15</b>
7.1 Implementations with a Time Tagger interface . . . . .	15
7.1.1 Detailed Description . . . . .	15
7.2 All measurements and virtual channels . . . . .	15
7.2.1 Detailed Description . . . . .	16
7.2.2 Event counting . . . . .	17
7.2.2.1 Detailed Description . . . . .	17
7.2.3 Time histograms . . . . .	17
7.2.3.1 Detailed Description . . . . .	18
7.2.4 Fluorescence-lifetime imaging (FLIM) . . . . .	18
7.2.4.1 Detailed Description . . . . .	18
7.2.5 Phase & frequency analysis . . . . .	19
7.2.5.1 Detailed Description . . . . .	19
7.2.6 Time-tag-streaming . . . . .	19
7.2.6.1 Detailed Description . . . . .	20
7.2.7 Helper classes . . . . .	20
7.2.7.1 Detailed Description . . . . .	21
7.2.8 Virtual Channels . . . . .	21
7.2.8.1 Detailed Description . . . . .	21
<b>8 Namespace Documentation</b>	<b>23</b>
8.1 Experimental Namespace Reference . . . . .	23
8.1.1 Detailed Description . . . . .	23
<b>9 Class Documentation</b>	<b>25</b>
9.1 IteratorBase::AbortError Class Reference . . . . .	25
9.1.1 Detailed Description . . . . .	25
9.1.2 Constructor & Destructor Documentation . . . . .	26

9.1.2.1	<a href="#">AbortError()</a>	26
9.1.2.2	<a href="#">~AbortError()</a>	26
9.2	<a href="#">ChannelGate Struct Reference</a>	26
9.2.1	<a href="#">Constructor &amp; Destructor Documentation</a>	26
9.2.1.1	<a href="#">ChannelGate()</a>	26
9.2.2	<a href="#">Member Data Documentation</a>	26
9.2.2.1	<a href="#">gate_close_channel</a>	26
9.2.2.2	<a href="#">gate_open_channel</a>	27
9.2.2.3	<a href="#">initial</a>	27
9.3	<a href="#">Coincidence Class Reference</a>	27
9.3.1	<a href="#">Detailed Description</a>	30
9.3.2	<a href="#">Constructor &amp; Destructor Documentation</a>	30
9.3.2.1	<a href="#">Coincidence()</a>	30
9.3.3	<a href="#">Member Function Documentation</a>	30
9.3.3.1	<a href="#">getChannel()</a>	30
9.4	<a href="#">Coincidences Class Reference</a>	31
9.4.1	<a href="#">Detailed Description</a>	33
9.4.2	<a href="#">Constructor &amp; Destructor Documentation</a>	33
9.4.2.1	<a href="#">Coincidences()</a>	33
9.4.2.2	<a href="#">~Coincidences()</a>	34
9.4.3	<a href="#">Member Function Documentation</a>	34
9.4.3.1	<a href="#">getChannels()</a>	34
9.4.3.2	<a href="#">next_impl()</a>	34
9.4.3.3	<a href="#">setCoincidenceWindow()</a>	34
9.5	<a href="#">Combinations Class Reference</a>	35
9.5.1	<a href="#">Detailed Description</a>	37
9.5.2	<a href="#">Constructor &amp; Destructor Documentation</a>	37
9.5.2.1	<a href="#">Combinations()</a>	37
9.5.2.2	<a href="#">~Combinations()</a>	38
9.5.3	<a href="#">Member Function Documentation</a>	38
9.5.3.1	<a href="#">clear_impl()</a>	38
9.5.3.2	<a href="#">getChannel()</a>	38
9.5.3.3	<a href="#">getCombination()</a>	38
9.5.3.4	<a href="#">getSumChannel()</a>	38
9.5.3.5	<a href="#">next_impl()</a>	38
9.6	<a href="#">Combiner Class Reference</a>	39
9.6.1	<a href="#">Detailed Description</a>	41
9.6.2	<a href="#">Constructor &amp; Destructor Documentation</a>	42
9.6.2.1	<a href="#">Combiner()</a>	42
9.6.2.2	<a href="#">~Combiner()</a>	42
9.6.3	<a href="#">Member Function Documentation</a>	42
9.6.3.1	<a href="#">clear_impl()</a>	42

9.6.3.2 getChannel()	42
9.6.3.3 getChannelCounts()	42
9.6.3.4 getData()	43
9.6.3.5 next_impl()	43
9.7 ConstantFractionDiscriminator Class Reference	43
9.7.1 Detailed Description	46
9.7.2 Constructor & Destructor Documentation	46
9.7.2.1 ConstantFractionDiscriminator()	46
9.7.2.2 ~ConstantFractionDiscriminator()	46
9.7.3 Member Function Documentation	46
9.7.3.1 getChannels()	46
9.7.3.2 next_impl()	47
9.7.3.3 on_start()	47
9.8 Correlation Class Reference	47
9.8.1 Detailed Description	50
9.8.2 Constructor & Destructor Documentation	50
9.8.2.1 Correlation()	50
9.8.2.2 ~Correlation()	51
9.8.3 Member Function Documentation	51
9.8.3.1 clear_impl()	51
9.8.3.2 getData()	51
9.8.3.3 getDataNormalized()	51
9.8.3.4 getIndex()	52
9.8.3.5 next_impl()	52
9.9 CountBetweenMarkers Class Reference	52
9.9.1 Detailed Description	55
9.9.2 Constructor & Destructor Documentation	55
9.9.2.1 CountBetweenMarkers()	55
9.9.2.2 ~CountBetweenMarkers()	56
9.9.3 Member Function Documentation	56
9.9.3.1 clear_impl()	56
9.9.3.2 getBinWidths()	56
9.9.3.3 getData()	56
9.9.3.4 getIndex()	56
9.9.3.5 next_impl()	56
9.9.3.6 ready()	57
9.10 Counter Class Reference	57
9.10.1 Detailed Description	60
9.10.2 Constructor & Destructor Documentation	60
9.10.2.1 Counter()	60
9.10.2.2 ~Counter()	60
9.10.3 Member Function Documentation	61

9.10.3.1 clear_impl()	61
9.10.3.2 getData()	61
9.10.3.3 getDataNormalized()	61
9.10.3.4 getDataObject()	61
9.10.3.5 getDataTotalCounts()	62
9.10.3.6 getIndex()	62
9.10.3.7 next_impl()	62
9.10.3.8 on_start()	63
9.11 CounterData Class Reference	63
9.11.1 Detailed Description	64
9.11.2 Constructor & Destructor Documentation	64
9.11.2.1 ~CounterData()	64
9.11.3 Member Function Documentation	64
9.11.3.1 getChannels()	64
9.11.3.2 getData()	64
9.11.3.3 getDataNormalized()	64
9.11.3.4 getDataTotalCounts()	64
9.11.3.5 getFrequency()	65
9.11.3.6 getIndex()	65
9.11.3.7 getOverflowMask()	65
9.11.3.8 getTime()	65
9.11.4 Member Data Documentation	65
9.11.4.1 dropped_bins	65
9.11.4.2 overflow	65
9.11.4.3 size	66
9.12 Countrate Class Reference	66
9.12.1 Detailed Description	68
9.12.2 Constructor & Destructor Documentation	69
9.12.2.1 Countrate()	69
9.12.2.2 ~Countrate()	69
9.12.3 Member Function Documentation	69
9.12.3.1 clear_impl()	69
9.12.3.2 getCountsTotal()	69
9.12.3.3 getData()	69
9.12.3.4 next_impl()	70
9.12.3.5 on_start()	70
9.13 CustomLogger Class Reference	70
9.13.1 Detailed Description	71
9.13.2 Constructor & Destructor Documentation	71
9.13.2.1 CustomLogger()	71
9.13.2.2 ~CustomLogger()	71
9.13.3 Member Function Documentation	71

9.13.3.1 disable()	71
9.13.3.2 enable()	71
9.13.3.3 Log()	71
9.14 CustomMeasurementBase Class Reference	71
9.14.1 Detailed Description	73
9.14.2 Constructor & Destructor Documentation	74
9.14.2.1 CustomMeasurementBase()	74
9.14.2.2 ~CustomMeasurementBase()	74
9.14.3 Member Function Documentation	74
9.14.3.1 _lock()	74
9.14.3.2 _unlock()	74
9.14.3.3 clear_impl()	74
9.14.3.4 finalize_init()	74
9.14.3.5 is_running()	74
9.14.3.6 next_impl()	74
9.14.3.7 next_impl_cs()	75
9.14.3.8 on_start()	75
9.14.3.9 on_stop()	75
9.14.3.10 register_channel()	75
9.14.3.11 stop_all_custom_measurements()	75
9.14.3.12 unregister_channel()	76
9.15 DelayedChannel Class Reference	76
9.15.1 Detailed Description	78
9.15.2 Constructor & Destructor Documentation	79
9.15.2.1 DelayedChannel() [1/2]	79
9.15.2.2 DelayedChannel() [2/2]	80
9.15.2.3 ~DelayedChannel()	80
9.15.3 Member Function Documentation	80
9.15.3.1 getChannel()	80
9.15.3.2 getChannels()	80
9.15.3.3 next_impl()	81
9.15.3.4 on_start()	81
9.15.3.5 setDelay()	81
9.16 Experimental::DlsSignalGenerator Class Reference	82
9.16.1 Constructor & Destructor Documentation	85
9.16.1.1 DlsSignalGenerator() [1/2]	85
9.16.1.2 DlsSignalGenerator() [2/2]	85
9.16.1.3 ~DlsSignalGenerator()	85
9.16.2 Member Function Documentation	85
9.16.2.1 get_intensity()	85
9.16.2.2 get_N()	85
9.17 Dump Class Reference	86

9.17.1 Detailed Description	88
9.17.2 Constructor & Destructor Documentation	88
9.17.2.1 Dump()	88
9.17.2.2 ~Dump()	88
9.17.3 Member Function Documentation	88
9.17.3.1 clear_impl()	88
9.17.3.2 next_impl()	88
9.17.3.3 on_start()	89
9.17.3.4 on_stop()	89
9.18 Event Struct Reference	89
9.18.1 Detailed Description	90
9.18.2 Member Data Documentation	90
9.18.2.1 state	90
9.18.2.2 time	90
9.19 EventGenerator Class Reference	90
9.19.1 Detailed Description	92
9.19.2 Constructor & Destructor Documentation	93
9.19.2.1 EventGenerator()	93
9.19.2.2 ~EventGenerator()	93
9.19.3 Member Function Documentation	93
9.19.3.1 clear_impl()	93
9.19.3.2 getChannel()	93
9.19.3.3 next_impl()	94
9.19.3.4 on_start()	94
9.20 Experimental::ExponentialSignalGenerator Class Reference	94
9.20.1 Constructor & Destructor Documentation	97
9.20.1.1 ExponentialSignalGenerator()	97
9.20.1.2 ~ExponentialSignalGenerator()	97
9.20.2 Member Function Documentation	98
9.20.2.1 get_next()	98
9.20.2.2 initialize()	98
9.20.2.3 on_restart()	98
9.21 FastBinning Class Reference	98
9.21.1 Detailed Description	98
9.21.2 Member Enumeration Documentation	98
9.21.2.1 Mode	98
9.21.3 Constructor & Destructor Documentation	99
9.21.3.1 FastBinning() [1/2]	99
9.21.3.2 FastBinning() [2/2]	99
9.21.4 Member Function Documentation	99
9.21.4.1 divide()	99
9.21.4.2 getMode()	99



9.22 Experimental::FcsSignalGenerator Class Reference	100
9.22.1 Constructor & Destructor Documentation	103
9.22.1.1 FcsSignalGenerator()	103
9.22.1.2 ~FcsSignalGenerator()	103
9.22.2 Member Function Documentation	103
9.22.2.1 get_intensity()	103
9.22.2.2 get_N()	103
9.22.2.3 set_boundary_limit()	103
9.23 FileReader Class Reference	104
9.23.1 Detailed Description	104
9.23.2 Constructor & Destructor Documentation	104
9.23.2.1 FileReader() [1/2]	104
9.23.2.2 FileReader() [2/2]	105
9.23.2.3 ~FileReader()	105
9.23.3 Member Function Documentation	105
9.23.3.1 getChannelList()	105
9.23.3.2 getConfiguration()	105
9.23.3.3 getData()	105
9.23.3.4 getDataRaw()	106
9.23.3.5 getLastMarker()	106
9.23.3.6 hasData()	106
9.24 FileWriter Class Reference	107
9.24.1 Detailed Description	109
9.24.2 Constructor & Destructor Documentation	109
9.24.2.1 FileWriter()	109
9.24.2.2 ~FileWriter()	109
9.24.3 Member Function Documentation	109
9.24.3.1 clear_impl()	109
9.24.3.2 getMaxFileSize()	110
9.24.3.3 getTotalEvents()	110
9.24.3.4 getTotalSize()	110
9.24.3.5 next_impl()	110
9.24.3.6 on_start()	111
9.24.3.7 on_stop()	111
9.24.3.8 setMarker()	111
9.24.3.9 setMaxFileSize()	111
9.24.3.10 split()	112
9.25 Flim Class Reference	112
9.25.1 Detailed Description	116
9.25.2 Constructor & Destructor Documentation	117
9.25.2.1 Flim()	117
9.25.2.2 ~Flim()	118

9.25.3 Member Function Documentation	118
9.25.3.1 clear_impl()	118
9.25.3.2 frameReady()	118
9.25.3.3 get_ready_index()	118
9.25.3.4 getCurrentFrame()	119
9.25.3.5 getCurrentFrameEx()	119
9.25.3.6 getCurrentFrameIntensity()	119
9.25.3.7 getFramesAcquired()	119
9.25.3.8 getIndex()	119
9.25.3.9 getReadyFrame()	119
9.25.3.10 getReadyFrameEx()	120
9.25.3.11 getReadyFrameIntensity()	120
9.25.3.12 getSummedFrames()	120
9.25.3.13 getSummedFramesEx()	122
9.25.3.14 getSummedFramesIntensity()	122
9.25.3.15 initialize()	122
9.25.3.16 on_frame_end()	123
9.25.4 Member Data Documentation	123
9.25.4.1 accum_diffs	123
9.25.4.2 back_frames	123
9.25.4.3 captured_frames	123
9.25.4.4 frame_begins	123
9.25.4.5 frame_ends	123
9.25.4.6 last_frame	123
9.25.4.7 pixels_completed	123
9.25.4.8 summed_frames	123
9.25.4.9 swap_chain_lock	124
9.25.4.10 total_frames	124
9.26 FlimAbstract Class Reference	124
9.26.1 Detailed Description	127
9.26.2 Constructor & Destructor Documentation	127
9.26.2.1 FlimAbstract()	127
9.26.2.2 ~FlimAbstract()	128
9.26.3 Member Function Documentation	128
9.26.3.1 clear_impl()	128
9.26.3.2 isAcquiring()	128
9.26.3.3 next_impl()	128
9.26.3.4 on_frame_end()	129
9.26.3.5 on_start()	129
9.26.3.6 process_tags()	129
9.26.4 Member Data Documentation	129
9.26.4.1 acquiring	129

9.26.4.2 acquisition_lock . . . . .	129
9.26.4.3 binner . . . . .	130
9.26.4.4 binwidth . . . . .	130
9.26.4.5 click_channel . . . . .	130
9.26.4.6 current_frame_begin . . . . .	130
9.26.4.7 current_frame_end . . . . .	130
9.26.4.8 data_base . . . . .	130
9.26.4.9 finish_after_outputframe . . . . .	130
9.26.4.10 frame . . . . .	130
9.26.4.11 frame_acquisition . . . . .	130
9.26.4.12 frame_begin_channel . . . . .	130
9.26.4.13 frames_completed . . . . .	131
9.26.4.14 initialized . . . . .	131
9.26.4.15 n_bins . . . . .	131
9.26.4.16 n_frame_average . . . . .	131
9.26.4.17 n_pixels . . . . .	131
9.26.4.18 pixel_acquisition . . . . .	131
9.26.4.19 pixel_begin_channel . . . . .	131
9.26.4.20 pixel_begins . . . . .	131
9.26.4.21 pixel_end_channel . . . . .	131
9.26.4.22 pixel_ends . . . . .	131
9.26.4.23 pixels_processed . . . . .	132
9.26.4.24 previous_starts . . . . .	132
9.26.4.25 start_channel . . . . .	132
9.26.4.26 ticks . . . . .	132
9.26.4.27 time_window . . . . .	132
9.27 FlimBase Class Reference . . . . .	132
9.27.1 Detailed Description . . . . .	135
9.27.2 Constructor & Destructor Documentation . . . . .	136
9.27.2.1 FlimBase() . . . . .	136
9.27.2.2 ~FlimBase() . . . . .	136
9.27.3 Member Function Documentation . . . . .	136
9.27.3.1 frameReady() . . . . .	136
9.27.3.2 initialize() . . . . .	137
9.27.3.3 on_frame_end() . . . . .	137
9.27.4 Member Data Documentation . . . . .	137
9.27.4.1 total_frames . . . . .	137
9.28 FlimFrameInfo Class Reference . . . . .	137
9.28.1 Detailed Description . . . . .	138
9.28.2 Constructor & Destructor Documentation . . . . .	138
9.28.2.1 ~FlimFrameInfo() . . . . .	138
9.28.3 Member Function Documentation . . . . .	138

9.28.3.1	<a href="#">getFrameNumber()</a>	138
9.28.3.2	<a href="#">getHistograms()</a>	138
9.28.3.3	<a href="#">getIntensities()</a>	138
9.28.3.4	<a href="#">getPixelBegins()</a>	138
9.28.3.5	<a href="#">getPixelEnds()</a>	139
9.28.3.6	<a href="#">getPixelPosition()</a>	139
9.28.3.7	<a href="#">getSummedCounts()</a>	139
9.28.3.8	<a href="#">isValid()</a>	139
9.28.4	<a href="#">Member Data Documentation</a>	139
9.28.4.1	<a href="#">bins</a>	139
9.28.4.2	<a href="#">frame_number</a>	139
9.28.4.3	<a href="#">pixel_position</a>	139
9.28.4.4	<a href="#">pixels</a>	140
9.28.4.5	<a href="#">valid</a>	140
9.29	<a href="#">FrequencyCounter Class Reference</a>	140
9.29.1	<a href="#">Detailed Description</a>	142
9.29.2	<a href="#">Constructor &amp; Destructor Documentation</a>	142
9.29.2.1	<a href="#">FrequencyCounter()</a>	142
9.29.2.2	<a href="#">~FrequencyCounter()</a>	143
9.29.3	<a href="#">Member Function Documentation</a>	143
9.29.3.1	<a href="#">clear_impl()</a>	143
9.29.3.2	<a href="#">getDataObject()</a>	143
9.29.3.3	<a href="#">next_impl()</a>	143
9.29.3.4	<a href="#">on_start()</a>	144
9.30	<a href="#">FrequencyCounterData Class Reference</a>	144
9.30.1	<a href="#">Constructor &amp; Destructor Documentation</a>	145
9.30.1.1	<a href="#">~FrequencyCounterData()</a>	145
9.30.2	<a href="#">Member Function Documentation</a>	145
9.30.2.1	<a href="#">getFrequency()</a>	145
9.30.2.2	<a href="#">getFrequencyInstantaneous()</a>	145
9.30.2.3	<a href="#">getIndex()</a>	145
9.30.2.4	<a href="#">getOverflowMask()</a>	145
9.30.2.5	<a href="#">getPeriodsCount()</a>	145
9.30.2.6	<a href="#">getPeriodsFraction()</a>	146
9.30.2.7	<a href="#">getPhase()</a>	146
9.30.2.8	<a href="#">getTime()</a>	146
9.30.3	<a href="#">Member Data Documentation</a>	146
9.30.3.1	<a href="#">align_to_reference</a>	146
9.30.3.2	<a href="#">channels_last_dim</a>	146
9.30.3.3	<a href="#">overflow_samples</a>	146
9.30.3.4	<a href="#">sample_offset</a>	146
9.30.3.5	<a href="#">sampling_interval</a>	147

9.30.3.6 size	147
9.31 FrequencyMultiplier Class Reference	147
9.31.1 Detailed Description	149
9.31.2 Constructor & Destructor Documentation	150
9.31.2.1 FrequencyMultiplier()	150
9.31.2.2 ~FrequencyMultiplier()	150
9.31.3 Member Function Documentation	150
9.31.3.1 clear_impl()	150
9.31.3.2 getChannel()	150
9.31.3.3 getMultiplier()	150
9.31.3.4 next_impl()	150
9.32 FrequencyStability Class Reference	151
9.32.1 Detailed Description	153
9.32.2 Constructor & Destructor Documentation	154
9.32.2.1 FrequencyStability()	154
9.32.2.2 ~FrequencyStability()	154
9.32.3 Member Function Documentation	155
9.32.3.1 clear_impl()	155
9.32.3.2 getDataObject()	155
9.32.3.3 next_impl()	155
9.32.3.4 on_start()	155
9.33 FrequencyStabilityData Class Reference	156
9.33.1 Detailed Description	156
9.33.2 Constructor & Destructor Documentation	157
9.33.2.1 ~FrequencyStabilityData()	157
9.33.3 Member Function Documentation	157
9.33.3.1 getADEV()	157
9.33.3.2 getADEVScaled()	157
9.33.3.3 getHDEV()	157
9.33.3.4 getHDEVScaled()	157
9.33.3.5 getMDEV()	157
9.33.3.6 getSTDD()	158
9.33.3.7 getTau()	158
9.33.3.8 getTDEV()	158
9.33.3.9 getTraceFrequency()	158
9.33.3.10 getTraceFrequencyAbsolute()	158
9.33.3.11 getTraceIndex()	159
9.33.3.12 getTracePhase()	159
9.34 Experimental::GammaSignalGenerator Class Reference	159
9.34.1 Constructor & Destructor Documentation	162
9.34.1.1 GammaSignalGenerator()	162
9.34.1.2 ~GammaSignalGenerator()	162

9.34.2 Member Function Documentation	162
9.34.2.1 get_next()	162
9.34.2.2 initialize()	162
9.34.2.3 on_restart()	162
9.35 GatedChannel Class Reference	163
9.35.1 Detailed Description	165
9.35.2 Constructor & Destructor Documentation	165
9.35.2.1 GatedChannel()	165
9.35.2.2 ~GatedChannel()	165
9.35.3 Member Function Documentation	166
9.35.3.1 clear_impl()	166
9.35.3.2 getChannel()	166
9.35.3.3 next_impl()	166
9.36 Experimental::GaussianSignalGenerator Class Reference	167
9.36.1 Constructor & Destructor Documentation	169
9.36.1.1 GaussianSignalGenerator()	169
9.36.1.2 ~GaussianSignalGenerator()	170
9.36.2 Member Function Documentation	170
9.36.2.1 get_next()	170
9.36.2.2 initialize()	170
9.36.2.3 on_restart()	170
9.37 Histogram Class Reference	170
9.37.1 Detailed Description	173
9.37.2 Constructor & Destructor Documentation	173
9.37.2.1 Histogram()	173
9.37.2.2 ~Histogram()	174
9.37.3 Member Function Documentation	174
9.37.3.1 clear_impl()	174
9.37.3.2 getData()	174
9.37.3.3 getIndex()	174
9.37.3.4 next_impl()	174
9.37.3.5 on_start()	175
9.38 Histogram2D Class Reference	175
9.38.1 Detailed Description	177
9.38.2 Constructor & Destructor Documentation	178
9.38.2.1 Histogram2D()	178
9.38.2.2 ~Histogram2D()	178
9.38.3 Member Function Documentation	178
9.38.3.1 clear_impl()	178
9.38.3.2 getData()	179
9.38.3.3 getIndex()	179
9.38.3.4 getIndex_1()	179

9.38.3.5 getIndex_2()	179
9.38.3.6 next_impl()	179
9.39 HistogramLogBins Class Reference	180
9.39.1 Detailed Description	182
9.39.2 Constructor & Destructor Documentation	183
9.39.2.1 HistogramLogBins()	183
9.39.2.2 ~HistogramLogBins()	184
9.39.3 Member Function Documentation	184
9.39.3.1 clear_impl()	184
9.39.3.2 getBinEdges()	184
9.39.3.3 getData()	184
9.39.3.4 getDataNormalizedCountsPerPs()	184
9.39.3.5 getDataNormalizedG2()	185
9.39.3.6 getDataObject()	185
9.39.3.7 next_impl()	185
9.40 HistogramLogBinsData Class Reference	185
9.40.1 Detailed Description	186
9.40.2 Constructor & Destructor Documentation	186
9.40.2.1 ~HistogramLogBinsData()	186
9.40.3 Member Function Documentation	186
9.40.3.1 getCounts()	186
9.40.3.2 getG2()	186
9.40.3.3 getG2Normalization()	187
9.40.4 Member Data Documentation	187
9.40.4.1 accumulation_time_click	187
9.40.4.2 accumulation_time_start	187
9.41 HistogramND Class Reference	187
9.41.1 Detailed Description	189
9.41.2 Constructor & Destructor Documentation	189
9.41.2.1 HistogramND()	189
9.41.2.2 ~HistogramND()	190
9.41.3 Member Function Documentation	190
9.41.3.1 clear_impl()	190
9.41.3.2 getData()	190
9.41.3.3 getIndex()	190
9.41.3.4 next_impl()	190
9.42 HistogramNDImpl< T > Class Template Reference	191
9.43 Iterator Class Reference	191
9.43.1 Detailed Description	194
9.43.2 Constructor & Destructor Documentation	194
9.43.2.1 Iterator()	194
9.43.2.2 ~Iterator()	194

9.43.3 Member Function Documentation	194
9.43.3.1 clear_impl()	194
9.43.3.2 next()	194
9.43.3.3 next_impl()	195
9.43.3.4 size()	195
9.44 IteratorBase Class Reference	195
9.44.1 Detailed Description	198
9.44.2 Constructor & Destructor Documentation	198
9.44.2.1 IteratorBase()	198
9.44.2.2 ~IteratorBase()	198
9.44.3 Member Function Documentation	199
9.44.3.1 abort()	199
9.44.3.2 checkForAbort() [1/2]	199
9.44.3.3 checkForAbort() [2/2]	199
9.44.3.4 clear()	199
9.44.3.5 clear_impl()	199
9.44.3.6 finish_running()	200
9.44.3.7 finishInitialization()	200
9.44.3.8 getCaptureDuration()	200
9.44.3.9 getConfiguration()	200
9.44.3.10 getLock()	200
9.44.3.11 getNewVirtualChannel()	201
9.44.3.12 isRunning()	201
9.44.3.13 lock()	201
9.44.3.14 next_impl()	201
9.44.3.15 on_start()	202
9.44.3.16 on_stop()	202
9.44.3.17 parallelize()	202
9.44.3.18 registerChannel()	202
9.44.3.19 start()	203
9.44.3.20 startFor()	203
9.44.3.21 stop()	203
9.44.3.22 unlock()	203
9.44.3.23 unregisterChannel()	203
9.44.3.24 waitUntilFinished()	204
9.44.4 Member Data Documentation	204
9.44.4.1 aborting	204
9.44.4.2 autostart	204
9.44.4.3 capture_duration	204
9.44.4.4 channels_registered	204
9.44.4.5 pre_capture_duration	205
9.44.4.6 running	205



9.44.4.7 tagger	205
9.45 Experimental::MarkovProcessGenerator Class Reference	205
9.45.1 Constructor & Destructor Documentation	207
9.45.1.1 MarkovProcessGenerator()	207
9.45.1.2 ~MarkovProcessGenerator()	208
9.45.2 Member Function Documentation	208
9.45.2.1 getChannel()	208
9.45.2.2 getChannels()	208
9.45.2.3 next_impl()	208
9.45.2.4 on_stop()	208
9.46 OrderedBarrier Class Reference	209
9.46.1 Detailed Description	209
9.46.2 Constructor & Destructor Documentation	209
9.46.2.1 OrderedBarrier()	209
9.46.2.2 ~OrderedBarrier()	209
9.46.3 Member Function Documentation	209
9.46.3.1 queue()	209
9.46.3.2 waitUntilFinished()	209
9.47 OrderedPipeline Class Reference	210
9.47.1 Detailed Description	210
9.47.2 Constructor & Destructor Documentation	210
9.47.2.1 OrderedPipeline()	210
9.47.2.2 ~OrderedPipeline()	210
9.48 OrderedBarrier::OrderInstance Class Reference	210
9.48.1 Detailed Description	211
9.48.2 Constructor & Destructor Documentation	211
9.48.2.1 OrderInstance() [1/2]	211
9.48.2.2 OrderInstance() [2/2]	211
9.48.2.3 ~OrderInstance()	211
9.48.3 Member Function Documentation	211
9.48.3.1 release()	211
9.48.3.2 sync()	211
9.49 Experimental::OscillatorSimulation Class Reference	212
9.49.1 Constructor & Destructor Documentation	214
9.49.1.1 OscillatorSimulation()	214
9.49.1.2 ~OscillatorSimulation()	215
9.49.2 Member Function Documentation	215
9.49.2.1 get_next()	215
9.49.2.2 initialize()	215
9.49.2.3 on_restart()	215
9.50 Experimental::PatternSignalGenerator Class Reference	216
9.50.1 Constructor & Destructor Documentation	218

9.50.1.1 PatternSignalGenerator()	218
9.50.1.2 ~PatternSignalGenerator()	219
9.50.2 Member Function Documentation	219
9.50.2.1 get_next()	219
9.50.2.2 initialize()	219
9.50.2.3 on_restart()	219
9.51 Experimental::PhotonGenerator Class Reference	219
9.51.1 Constructor & Destructor Documentation	222
9.51.1.1 PhotonGenerator()	222
9.51.1.2 ~PhotonGenerator()	222
9.51.2 Member Function Documentation	222
9.51.2.1 finalize_init()	222
9.51.2.2 get_intensity()	223
9.51.2.3 get_next()	223
9.51.2.4 get_T_PERIOD()	223
9.51.2.5 initialize()	223
9.51.2.6 on_restart()	223
9.51.2.7 set_T_PERIOD()	223
9.51.3 Member Data Documentation	223
9.51.3.1 T_PERIOD	223
9.52 Experimental::PhotonNumber Class Reference	224
9.52.1 Detailed Description	226
9.52.2 Constructor & Destructor Documentation	226
9.52.2.1 PhotonNumber()	226
9.52.2.2 ~PhotonNumber()	226
9.52.3 Member Function Documentation	226
9.52.3.1 clear_impl()	226
9.52.3.2 getChannels()	227
9.52.3.3 next_impl()	227
9.53 Experimental::PulsePerSecondData Class Reference	227
9.53.1 Detailed Description	228
9.53.2 Constructor & Destructor Documentation	228
9.53.2.1 ~PulsePerSecondData()	228
9.53.3 Member Function Documentation	228
9.53.3.1 getIndices()	228
9.53.3.2 getReferenceOffsets()	229
9.53.3.3 getSignalOffsets()	229
9.53.3.4 getStatus()	229
9.53.3.5 getUtcDates()	229
9.53.3.6 getUtcSeconds()	229
9.53.4 Member Data Documentation	229
9.53.4.1 size	229

9.54 Experimental::PulsePerSecondMonitor Class Reference	230
9.54.1 Detailed Description	232
9.54.2 Constructor & Destructor Documentation	232
9.54.2.1 PulsePerSecondMonitor()	232
9.54.2.2 ~PulsePerSecondMonitor()	232
9.54.3 Member Function Documentation	232
9.54.3.1 clear_impl()	232
9.54.3.2 getDataObject()	233
9.54.3.3 next_impl()	233
9.54.3.4 on_start()	233
9.55 Sampler Class Reference	234
9.55.1 Detailed Description	236
9.55.2 Constructor & Destructor Documentation	236
9.55.2.1 Sampler()	236
9.55.2.2 ~Sampler()	237
9.55.3 Member Function Documentation	237
9.55.3.1 clear_impl()	237
9.55.3.2 getData()	237
9.55.3.3 getDataAsMask()	237
9.55.3.4 next_impl()	237
9.55.3.5 on_start()	238
9.56 Scope Class Reference	238
9.56.1 Detailed Description	241
9.56.2 Constructor & Destructor Documentation	241
9.56.2.1 Scope()	241
9.56.2.2 ~Scope()	242
9.56.3 Member Function Documentation	242
9.56.3.1 clear_impl()	242
9.56.3.2 getData()	242
9.56.3.3 getWindowSize()	242
9.56.3.4 next_impl()	242
9.56.3.5 ready()	243
9.56.3.6 triggered()	243
9.57 Experimental::SignalGeneratorBase Class Reference	243
9.57.1 Constructor & Destructor Documentation	245
9.57.1.1 SignalGeneratorBase()	245
9.57.1.2 ~SignalGeneratorBase()	245
9.57.2 Member Function Documentation	246
9.57.2.1 get_next()	246
9.57.2.2 getChannel()	246
9.57.2.3 initialize()	246
9.57.2.4 isProcessingFinished()	246

9.57.2.5 next_impl()	246
9.57.2.6 on_restart()	247
9.57.2.7 on_stop()	247
9.57.2.8 set_processing_finished()	247
9.57.3 Member Data Documentation	247
9.57.3.1 impl	247
9.58 Experimental::SimDetector Class Reference	247
9.58.1 Constructor & Destructor Documentation	248
9.58.1.1 SimDetector()	248
9.58.1.2 ~SimDetector()	248
9.58.2 Member Function Documentation	248
9.58.2.1 getChannel()	248
9.59 Experimental::SimLifetime Class Reference	249
9.59.1 Constructor & Destructor Documentation	251
9.59.1.1 SimLifetime()	251
9.59.1.2 ~SimLifetime()	251
9.59.2 Member Function Documentation	251
9.59.2.1 getChannel()	251
9.59.2.2 next_impl()	251
9.59.2.3 registerEmissionReactor()	252
9.59.2.4 registerLifetimeReactor()	252
9.60 Experimental::SimSignalSplitter Class Reference	252
9.60.1 Constructor & Destructor Documentation	254
9.60.1.1 SimSignalSplitter()	254
9.60.1.2 ~SimSignalSplitter()	255
9.60.2 Member Function Documentation	255
9.60.2.1 getChannels()	255
9.60.2.2 getLeftChannel()	255
9.60.2.3 getRightChannel()	255
9.60.2.4 next_impl()	255
9.61 SoftwareClockState Struct Reference	256
9.61.1 Member Data Documentation	256
9.61.1.1 averaging_periods	256
9.61.1.2 clock_period	256
9.61.1.3 enabled	256
9.61.1.4 error_counter	256
9.61.1.5 ideal_clock_channel	256
9.61.1.6 input_channel	256
9.61.1.7 is_locked	257
9.61.1.8 last_ideal_clock_event	257
9.61.1.9 period_error	257
9.61.1.10 phase_error_estimation	257

9.62 StartStop Class Reference . . . . .	257
9.62.1 Detailed Description . . . . .	259
9.62.2 Constructor & Destructor Documentation . . . . .	260
9.62.2.1 StartStop() . . . . .	260
9.62.2.2 ~StartStop() . . . . .	260
9.62.3 Member Function Documentation . . . . .	260
9.62.3.1 clear_impl() . . . . .	260
9.62.3.2 getData() . . . . .	260
9.62.3.3 next_impl() . . . . .	261
9.62.3.4 on_start() . . . . .	261
9.63 SynchronizedMeasurements Class Reference . . . . .	261
9.63.1 Detailed Description . . . . .	262
9.63.2 Constructor & Destructor Documentation . . . . .	262
9.63.2.1 SynchronizedMeasurements() . . . . .	262
9.63.2.2 ~SynchronizedMeasurements() . . . . .	263
9.63.3 Member Function Documentation . . . . .	263
9.63.3.1 clear() . . . . .	263
9.63.3.2 getTagger() . . . . .	263
9.63.3.3 isRunning() . . . . .	263
9.63.3.4 registerMeasurement() . . . . .	263
9.63.3.5 runCallback() . . . . .	263
9.63.3.6 start() . . . . .	264
9.63.3.7 startFor() . . . . .	264
9.63.3.8 stop() . . . . .	264
9.63.3.9 unregisterMeasurement() . . . . .	264
9.63.3.10 waitUntilFinished() . . . . .	264
9.64 SyntheticSingleTag Class Reference . . . . .	265
9.64.1 Detailed Description . . . . .	267
9.64.2 Constructor & Destructor Documentation . . . . .	267
9.64.2.1 SyntheticSingleTag() . . . . .	267
9.64.2.2 ~SyntheticSingleTag() . . . . .	267
9.64.3 Member Function Documentation . . . . .	267
9.64.3.1 getChannel() . . . . .	267
9.64.3.2 next_impl() . . . . .	267
9.64.3.3 trigger() . . . . .	268
9.65 Tag Struct Reference . . . . .	268
9.65.1 Detailed Description . . . . .	269
9.65.2 Member Enumeration Documentation . . . . .	269
9.65.2.1 Type . . . . .	269
9.65.3 Constructor & Destructor Documentation . . . . .	270
9.65.3.1 Tag() [1/3] . . . . .	270
9.65.3.2 Tag() [2/3] . . . . .	270

9.65.3.3 Tag() [ 3/3]	270
9.65.4 Member Data Documentation	270
9.65.4.1 channel	270
9.65.4.2 missed_events	270
9.65.4.3 reserved	270
9.65.4.4 time	271
9.65.4.5 TimeTag	271
9.66 TimeDifferences Class Reference	271
9.66.1 Detailed Description	273
9.66.2 Constructor & Destructor Documentation	274
9.66.2.1 TimeDifferences()	274
9.66.2.2 ~TimeDifferences()	275
9.66.3 Member Function Documentation	275
9.66.3.1 clear_impl()	275
9.66.3.2 getCounts()	275
9.66.3.3 getData()	275
9.66.3.4 getHistogramIndex()	275
9.66.3.5 getIndex()	275
9.66.3.6 next_impl()	275
9.66.3.7 on_start()	276
9.66.3.8 ready()	276
9.66.3.9 setMaxCounts()	276
9.67 TimeDifferencesImpl< T > Class Template Reference	276
9.68 TimeDifferencesND Class Reference	277
9.68.1 Detailed Description	279
9.68.2 Constructor & Destructor Documentation	280
9.68.2.1 TimeDifferencesND()	280
9.68.2.2 ~TimeDifferencesND()	280
9.68.3 Member Function Documentation	280
9.68.3.1 clear_impl()	280
9.68.3.2 getData()	281
9.68.3.3 getIndex()	281
9.68.3.4 next_impl()	281
9.68.3.5 on_start()	281
9.69 TimeTagger Class Reference	282
9.69.1 Detailed Description	287
9.69.2 Member Function Documentation	287
9.69.2.1 autoCalibration()	287
9.69.2.2 clearConditionalFilter()	287
9.69.2.3 disableFpgaLink()	287
9.69.2.4 disableLEDs()	287
9.69.2.5 enableFpgaLink()	287

9.69.2.6 factoryAccess()	288
9.69.2.7 getChannelList()	288
9.69.2.8 getChannelNumberScheme()	288
9.69.2.9 getConditionalFilterFiltered()	289
9.69.2.10 getConditionalFilterTrigger()	289
9.69.2.11 getDACRange()	289
9.69.2.12 getDeviceLicense()	289
9.69.2.13 getDistributionCount()	289
9.69.2.14 getDistributionPSEcs()	289
9.69.2.15 getEventDivider()	289
9.69.2.16 getFirmwareVersion()	290
9.69.2.17 getHardwareBufferSize()	290
9.69.2.18 getHardwareDelayCompensation()	290
9.69.2.19 getInputHysteresis()	291
9.69.2.20 getInputImpedanceHigh()	291
9.69.2.21 getInputMux()	291
9.69.2.22 getModel()	292
9.69.2.23 getNormalization()	292
9.69.2.24 getPcbVersion()	292
9.69.2.25 getPsPerClock()	292
9.69.2.26 getSensorData()	293
9.69.2.27 getSerial()	293
9.69.2.28 getStreamBlockSizeEvents()	293
9.69.2.29 getStreamBlockSizeLatency()	293
9.69.2.30 getTestSignalDivider()	293
9.69.2.31 getTriggerLevel()	293
9.69.2.32 isChannelRegistered()	293
9.69.2.33 isServerRunning()	293
9.69.2.34 reset()	294
9.69.2.35 setConditionalFilter()	294
9.69.2.36 setEventDivider()	294
9.69.2.37 setHardwareBufferSize()	295
9.69.2.38 setInputHysteresis()	295
9.69.2.39 setInputImpedanceHigh()	295
9.69.2.40 setInputMux()	296
9.69.2.41 setLED()	296
9.69.2.42 setNormalization()	296
9.69.2.43 setSoundFrequency()	296
9.69.2.44 setStreamBlockSize()	297
9.69.2.45 setTestSignalDivider()	297
9.69.2.46 setTimeTaggerNetworkStreamCompression()	297
9.69.2.47 setTriggerLevel()	298

9.69.2.48 startServer()	298
9.69.2.49 stopServer()	298
9.69.2.50 updateBMCFirmware()	298
9.69.2.51 xtra_getAuxOut()	299
9.69.2.52 xtra_getAuxOutSignalDivider()	299
9.69.2.53 xtra_getAuxOutSignalDutyCycle()	299
9.69.2.54 xtra_getAvgRisingFalling()	300
9.69.2.55 xtra_getClockAutoSelect()	300
9.69.2.56 xtra_getClockSource()	300
9.69.2.57 xtra_getHighPrioChannel()	300
9.69.2.58 xtra_measureTriggerLevel()	301
9.69.2.59 xtra_setAuxOut()	301
9.69.2.60 xtra_setAuxOutSignal()	301
9.69.2.61 xtra_setAvgRisingFalling()	302
9.69.2.62 xtra_setClockAutoSelect()	302
9.69.2.63 xtra_setClockOut()	303
9.69.2.64 xtra_setClockSource()	303
9.69.2.65 xtra_setFanSpeed()	303
9.69.2.66 xtra_setHighPrioChannel()	303
9.70 TimeTaggerBase Class Reference	304
9.70.1 Detailed Description	306
9.70.2 Member Typedef Documentation	306
9.70.2.1 IteratorCallback	306
9.70.2.2 IteratorCallbackMap	306
9.70.3 Constructor & Destructor Documentation	306
9.70.3.1 TimeTaggerBase() [1/2]	306
9.70.3.2 ~TimeTaggerBase()	307
9.70.3.3 TimeTaggerBase() [2/2]	307
9.70.4 Member Function Documentation	307
9.70.4.1 addChild()	307
9.70.4.2 addIterator()	307
9.70.4.3 clearOverflows()	307
9.70.4.4 disableSoftwareClock()	307
9.70.4.5 freeIterator()	307
9.70.4.6 freeVirtualChannel()	308
9.70.4.7 getConfiguration()	308
9.70.4.8 getDeadtime()	308
9.70.4.9 getDelayHardware()	308
9.70.4.10 getDelaySoftware()	309
9.70.4.11 getFence()	309
9.70.4.12 getInputDelay()	309
9.70.4.13 getInvertedChannel()	310



9.70.4.14	<a href="#">getNewVirtualChannel()</a>	310
9.70.4.15	<a href="#">getOverflows()</a>	310
9.70.4.16	<a href="#">getOverflowsAndClear()</a>	310
9.70.4.17	<a href="#">getSoftwareClockState()</a>	311
9.70.4.18	<a href="#">getTestSignal()</a>	311
9.70.4.19	<a href="#">isUnusedChannel()</a>	311
9.70.4.20	<a href="#">operator=()</a>	311
9.70.4.21	<a href="#">registerChannel()</a> [1/2]	311
9.70.4.22	<a href="#">registerChannel()</a> [2/2]	312
9.70.4.23	<a href="#">release()</a>	312
9.70.4.24	<a href="#">removeChild()</a>	312
9.70.4.25	<a href="#">runSynchronized()</a>	312
9.70.4.26	<a href="#">setDeadtime()</a>	312
9.70.4.27	<a href="#">setDelayHardware()</a>	313
9.70.4.28	<a href="#">setDelaySoftware()</a>	313
9.70.4.29	<a href="#">setInputDelay()</a>	313
9.70.4.30	<a href="#">setSoftwareClock()</a>	314
9.70.4.31	<a href="#">setTestSignal()</a> [1/2]	314
9.70.4.32	<a href="#">setTestSignal()</a> [2/2]	315
9.70.4.33	<a href="#">sync()</a>	315
9.70.4.34	<a href="#">unregisterChannel()</a> [1/2]	315
9.70.4.35	<a href="#">unregisterChannel()</a> [2/2]	316
9.70.4.36	<a href="#">waitForFence()</a>	316
9.71	<a href="#">TimeTaggerNetwork Class Reference</a>	316
9.71.1	<a href="#">Detailed Description</a>	320
9.71.2	<a href="#">Member Function Documentation</a>	320
9.71.2.1	<a href="#">clearConditionalFilter()</a>	320
9.71.2.2	<a href="#">clearOverflowsClient()</a>	320
9.71.2.3	<a href="#">getChannelList()</a>	321
9.71.2.4	<a href="#">getChannelNumberScheme()</a>	321
9.71.2.5	<a href="#">getConditionalFilterFiltered()</a>	321
9.71.2.6	<a href="#">getConditionalFilterTrigger()</a>	321
9.71.2.7	<a href="#">getDACRange()</a>	321
9.71.2.8	<a href="#">getDelayClient()</a>	321
9.71.2.9	<a href="#">getDeviceLicense()</a>	322
9.71.2.10	<a href="#">getEventDivider()</a>	322
9.71.2.11	<a href="#">getFirmwareVersion()</a>	322
9.71.2.12	<a href="#">getHardwareBufferSize()</a>	323
9.71.2.13	<a href="#">getHardwareDelayCompensation()</a>	323
9.71.2.14	<a href="#">getInputHysteresis()</a>	323
9.71.2.15	<a href="#">getInputImpedanceHigh()</a>	323
9.71.2.16	<a href="#">getModel()</a>	324

9.71.2.17	getNormalization()	324
9.71.2.18	getOverflowsAndClearClient()	324
9.71.2.19	getOverflowsClient()	324
9.71.2.20	getPcbVersion()	324
9.71.2.21	getPsPerClock()	325
9.71.2.22	getSensorData()	325
9.71.2.23	getSerial()	325
9.71.2.24	getStreamBlockSizeEvents()	325
9.71.2.25	getStreamBlockSizeLatency()	325
9.71.2.26	getTestSignal()	325
9.71.2.27	getTestSignalDivider()	325
9.71.2.28	getTriggerLevel()	326
9.71.2.29	isConnected()	326
9.71.2.30	setConditionalFilter()	326
9.71.2.31	setDelayClient()	326
9.71.2.32	setEventDivider()	327
9.71.2.33	setHardwareBufferSize()	327
9.71.2.34	setInputHysteresis()	327
9.71.2.35	setInputImpedanceHigh()	328
9.71.2.36	setLED()	328
9.71.2.37	setNormalization()	328
9.71.2.38	setSoundFrequency()	329
9.71.2.39	setStreamBlockSize()	329
9.71.2.40	setTestSignalDivider()	329
9.71.2.41	setTimeTaggerNetworkStreamCompression()	330
9.71.2.42	setTriggerLevel()	330
9.72	TimeTaggerVirtual Class Reference	330
9.72.1	Detailed Description	333
9.72.2	Member Function Documentation	333
9.72.2.1	clearConditionalFilter()	333
9.72.2.2	getChannelList()	333
9.72.2.3	getConditionalFilterFiltered()	333
9.72.2.4	getConditionalFilterTrigger()	333
9.72.2.5	getReplaySpeed()	334
9.72.2.6	replay()	334
9.72.2.7	reset()	334
9.72.2.8	setConditionalFilter()	335
9.72.2.9	setReplaySpeed()	335
9.72.2.10	stop()	335
9.72.2.11	waitForCompletion()	335
9.73	TimeTagStream Class Reference	336
9.73.1	Detailed Description	338

9.73.2 Constructor & Destructor Documentation	338
9.73.2.1 TimeTagStream()	338
9.73.2.2 ~TimeTagStream()	339
9.73.3 Member Function Documentation	339
9.73.3.1 clear_impl()	339
9.73.3.2 getCounts()	339
9.73.3.3 getData()	339
9.73.3.4 next_impl()	339
9.74 TimeTagStreamBuffer Class Reference	340
9.74.1 Detailed Description	340
9.74.2 Constructor & Destructor Documentation	340
9.74.2.1 ~TimeTagStreamBuffer()	340
9.74.3 Member Function Documentation	341
9.74.3.1 getChannels()	341
9.74.3.2 getEventTypes()	341
9.74.3.3 getMissedEvents()	341
9.74.3.4 getOverflows()	341
9.74.3.5 getTimestamps()	341
9.74.4 Member Data Documentation	341
9.74.4.1 hasOverflows	341
9.74.4.2 size	341
9.74.4.3 tGetData	341
9.74.4.4 tStart	342
9.75 Experimental::TransformCrosstalk Class Reference	342
9.75.1 Constructor & Destructor Documentation	344
9.75.1.1 TransformCrosstalk()	344
9.75.1.2 ~TransformCrosstalk()	345
9.75.2 Member Function Documentation	345
9.75.2.1 getChannel()	345
9.75.2.2 next_impl()	345
9.76 Experimental::TransformDeadtime Class Reference	345
9.76.1 Constructor & Destructor Documentation	348
9.76.1.1 TransformDeadtime()	348
9.76.1.2 ~TransformDeadtime()	348
9.76.2 Member Function Documentation	348
9.76.2.1 getChannel()	348
9.76.2.2 next_impl()	348
9.77 Experimental::TransformEfficiency Class Reference	349
9.77.1 Constructor & Destructor Documentation	351
9.77.1.1 TransformEfficiency()	351
9.77.1.2 ~TransformEfficiency()	352
9.77.2 Member Function Documentation	352

9.77.2.1	getChannel()	352
9.77.2.2	next_impl()	352
9.78	Experimental::TransformGaussianBroadening Class Reference	353
9.78.1	Constructor & Destructor Documentation	355
9.78.1.1	TransformGaussianBroadening()	355
9.78.1.2	~TransformGaussianBroadening()	355
9.78.2	Member Function Documentation	355
9.78.2.1	getChannel()	355
9.78.2.2	next_impl()	355
9.79	TriggerOnCountrate Class Reference	356
9.79.1	Detailed Description	359
9.79.2	Constructor & Destructor Documentation	359
9.79.2.1	TriggerOnCountrate()	359
9.79.2.2	~TriggerOnCountrate()	360
9.79.3	Member Function Documentation	360
9.79.3.1	clear_impl()	360
9.79.3.2	getChannelAbove()	360
9.79.3.3	getChannelBelow()	360
9.79.3.4	getChannels()	360
9.79.3.5	getCurrentCountrate()	361
9.79.3.6	injectCurrentState()	361
9.79.3.7	isAbove()	361
9.79.3.8	isBelow()	361
9.79.3.9	next_impl()	361
9.79.3.10	on_start()	362
9.80	Experimental::TwoStateExponentialSignalGenerator Class Reference	362
9.80.1	Constructor & Destructor Documentation	365
9.80.1.1	TwoStateExponentialSignalGenerator()	365
9.80.1.2	~TwoStateExponentialSignalGenerator()	365
9.80.2	Member Function Documentation	365
9.80.2.1	get_next()	365
9.80.2.2	initialize()	365
9.80.2.3	on_restart()	365
9.81	Experimental::UniformSignalGenerator Class Reference	366
9.81.1	Constructor & Destructor Documentation	368
9.81.1.1	UniformSignalGenerator()	368
9.81.1.2	~UniformSignalGenerator()	369
9.81.2	Member Function Documentation	369
9.81.2.1	get_next()	369
9.81.2.2	initialize()	369
9.81.2.3	on_restart()	369

<b>10 File Documentation</b>	<b>371</b>
10.1 Iterators.h File Reference	371
10.1.1 Macro Definition Documentation	374
10.1.1.1 BINNING_TEMPLATE_HELPER	374
10.1.2 Enumeration Type Documentation	375
10.1.2.1 CoincidenceTimestamp	375
10.1.2.2 GatedChannelInitial	375
10.1.2.3 State	375
10.2 Iterators.h	376
10.3 TimeTagger.h File Reference	398
10.3.1 Macro Definition Documentation	401
10.3.1.1 channel_t	401
10.3.1.2 ErrorLog	401
10.3.1.3 ErrorLogSuppressed	401
10.3.1.4 GET_DATA_1D	401
10.3.1.5 GET_DATA_1D_OP1	402
10.3.1.6 GET_DATA_1D_OP2	402
10.3.1.7 GET_DATA_2D	402
10.3.1.8 GET_DATA_2D_OP1	402
10.3.1.9 GET_DATA_2D_OP2	403
10.3.1.10 GET_DATA_3D	403
10.3.1.11 InfoLog	403
10.3.1.12 InfoLogSuppressed	403
10.3.1.13 LogMessage	403
10.3.1.14 LogMessageSuppressed	403
10.3.1.15 timestamp_t	404
10.3.1.16 TIMETAGGER_VERSION	404
10.3.1.17 TT_API	404
10.3.1.18 WarningLog	404
10.3.1.19 WarningLogSuppressed	404
10.3.2 Typedef Documentation	404
10.3.2.1 _Iterator	404
10.3.2.2 logger_callback	404
10.3.3 Enumeration Type Documentation	404
10.3.3.1 AccessMode	404
10.3.3.2 ChannelEdge	405
10.3.3.3 FpgaLinkInterface	405
10.3.3.4 FrontendType	405
10.3.3.5 LanguageUsed	406
10.3.3.6 LogLevel	406
10.3.3.7 Resolution	406
10.3.3.8 UsageStatisticsStatus	407

10.3.4 Function Documentation	407
10.3.4.1 checkSystemLibraries()	407
10.3.4.2 createTimeTagger()	407
10.3.4.3 createTimeTaggerNetwork()	407
10.3.4.4 createTimeTaggerVirtual()	408
10.3.4.5 extractDeviceLicense()	408
10.3.4.6 flashLicense()	408
10.3.4.7 freeTimeTagger()	408
10.3.4.8 getTimeTaggerChannelNumberScheme()	409
10.3.4.9 getTimeTaggerModel()	409
10.3.4.10 getTimeTaggerServerInfo()	409
10.3.4.11 getUsageStatisticsReport()	409
10.3.4.12 getUsageStatisticsStatus()	410
10.3.4.13 getVersion()	410
10.3.4.14 hasTimeTaggerVirtualLicense()	410
10.3.4.15 LogBase()	410
10.3.4.16 mergeStreamFiles()	410
10.3.4.17 operator==()	411
10.3.4.18 scanTimeTagger()	411
10.3.4.19 scanTimeTaggerServers()	411
10.3.4.20 setCustomBitFileName()	411
10.3.4.21 setFrontend()	412
10.3.4.22 setLanguageInfo()	412
10.3.4.23 setLogger()	412
10.3.4.24 setTimeTaggerChannelNumberScheme()	412
10.3.4.25 setUsageStatisticsStatus()	413
10.3.5 Variable Documentation	413
10.3.5.1 CHANNEL_UNUSED	413
10.3.5.2 CHANNEL_UNUSED_OLD	413
10.3.5.3 TT_CHANNEL_FALLING_EDGES	414
10.3.5.4 TT_CHANNEL_NUMBER_SCHEME_AUTO	414
10.3.5.5 TT_CHANNEL_NUMBER_SCHEME_DEFAULT	414
10.3.5.6 TT_CHANNEL_NUMBER_SCHEME_ONE	414
10.3.5.7 TT_CHANNEL_NUMBER_SCHEME_ZERO	414
10.3.5.8 TT_CHANNEL_RISING_AND_FALLING_EDGES	414
10.3.5.9 TT_CHANNEL_RISING_EDGES	414
10.4 TimeTagger.h	415

# Chapter 1

## TimeTagger

backend for [TimeTagger](#), an OpalKelly based single photon counting library

backend for [TimeTagger](#), an OpalKelly based single photon counting library

### Author

Markus Wick [markus@swabianinstruments.com](mailto:markus@swabianinstruments.com)

Helmut Fedder [helmut@swabianinstruments.com](mailto:helmut@swabianinstruments.com)

Michael Schlagmüller [michael@swabianinstruments.com](mailto:michael@swabianinstruments.com)

[TimeTagger](#) provides an easy to use and cost effective hardware solution for time-resolved single photon counting applications.

This document describes the C++ native interface to the [TimeTagger](#) device.





## Chapter 2

# Topic Index

### 2.1 Topics

Here is a list of all topics with brief descriptions:

Implementations with a Time Tagger interface . . . . .	15
All measurements and virtual channels . . . . .	15
Event counting . . . . .	17
Time histograms . . . . .	17
Fluorescence-lifetime imaging (FLIM) . . . . .	18
Phase & frequency analysis . . . . .	19
Time-tag-streaming . . . . .	19
Helper classes . . . . .	20
Virtual Channels . . . . .	21



## Chapter 3

# Namespace Index

### 3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

#### Experimental

Namespace for features, which are still in development and are likely to change . . . . . 23



## Chapter 4

# Hierarchical Index

### 4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

ChannelGate . . . . .	26
CounterData . . . . .	63
CustomLogger . . . . .	70
Event . . . . .	89
FastBinning . . . . .	98
FileReader . . . . .	104
FlimFrameInfo . . . . .	137
FrequencyCounterData . . . . .	144
FrequencyStabilityData . . . . .	156
HistogramLogBinsData . . . . .	185
HistogramNDImpl< T > . . . . .	191
IteratorBase . . . . .	195
Coincidences . . . . .	31
Coincidence . . . . .	27
Combinations . . . . .	35
Combiner . . . . .	39
ConstantFractionDiscriminator . . . . .	43
Correlation . . . . .	47
CountBetweenMarkers . . . . .	52
Counter . . . . .	57
Countrate . . . . .	66
CustomMeasurementBase . . . . .	71
DelayedChannel . . . . .	76
Dump . . . . .	86
EventGenerator . . . . .	90
Experimental::MarkovProcessGenerator . . . . .	205
Experimental::PhotonNumber . . . . .	224
Experimental::PulsePerSecondMonitor . . . . .	230
Experimental::SignalGeneratorBase . . . . .	243
Experimental::ExponentialSignalGenerator . . . . .	94
Experimental::GammaSignalGenerator . . . . .	159
Experimental::GaussianSignalGenerator . . . . .	167
Experimental::OscillatorSimulation . . . . .	212
Experimental::PatternSignalGenerator . . . . .	216
Experimental::PhotonGenerator . . . . .	219

Experimental::DlsSignalGenerator . . . . .	82
Experimental::FcsSignalGenerator . . . . .	100
Experimental::TwoStateExponentialSignalGenerator . . . . .	362
Experimental::UniformSignalGenerator . . . . .	366
Experimental::SimLifetime . . . . .	249
Experimental::SimSignalSplitter . . . . .	252
Experimental::TransformCrosstalk . . . . .	342
Experimental::TransformDeadtime . . . . .	345
Experimental::TransformEfficiency . . . . .	349
Experimental::TransformGaussianBroadening . . . . .	353
FileWriter . . . . .	107
FlimAbstract . . . . .	124
Flim . . . . .	112
FlimBase . . . . .	132
FrequencyCounter . . . . .	140
FrequencyMultiplier . . . . .	147
FrequencyStability . . . . .	151
GatedChannel . . . . .	163
Histogram . . . . .	170
Histogram2D . . . . .	175
HistogramLogBins . . . . .	180
HistogramND . . . . .	187
Iterator . . . . .	191
Sampler . . . . .	234
Scope . . . . .	238
StartStop . . . . .	257
SyntheticSingleTag . . . . .	265
TimeDifferences . . . . .	271
TimeDifferencesND . . . . .	277
TimeTagStream . . . . .	336
TriggerOnCountrate . . . . .	356
OrderedBarrier . . . . .	209
OrderedPipeline . . . . .	210
OrderedBarrier::OrderInstance . . . . .	210
Experimental::PulsePerSecondData . . . . .	227
std::runtime_error	
IteratorBase::AbortError . . . . .	25
Experimental::SimDetector . . . . .	247
SoftwareClockState . . . . .	256
SynchronizedMeasurements . . . . .	261
Tag . . . . .	268
TimeDifferencesImpl< T > . . . . .	276
TimeTaggerBase . . . . .	304
TimeTagger . . . . .	282
TimeTaggerNetwork . . . . .	316
TimeTaggerVirtual . . . . .	330
TimeTagStreamBuffer . . . . .	340

## Chapter 5

# Class Index

### 5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">IteratorBase::AbortError</a>	
A custom runtime error thrown by the <code>abort</code> call. This can be caught and handled by measurement classes, including <code>CustomMeasurement</code> , to perform actions within the abortion process	25
<a href="#">ChannelGate</a>	26
<a href="#">Coincidence</a>	
<a href="#">Coincidence</a> monitor for one channel group	27
<a href="#">Coincidences</a>	
<a href="#">Coincidence</a> monitor for many channel groups	31
<a href="#">Combinations</a>	35
<a href="#">Combiner</a>	
Combine some channels in a virtual channel which has a tick for each tick in the input channels	39
<a href="#">ConstantFractionDiscriminator</a>	
Virtual CFD implementation which returns the mean time between a rising and a falling pair of edges	43
<a href="#">Correlation</a>	
Auto- and Cross-correlation measurement	47
<a href="#">CountBetweenMarkers</a>	
Simple counter where external marker signals determine the bins	52
<a href="#">Counter</a>	
Simple counter on one or more channels	57
<a href="#">CounterData</a>	
Helper object as return value for <a href="#">Counter::getDataObject</a>	63
<a href="#">Countrate</a>	
Count rate on one or more channels	66
<a href="#">CustomLogger</a>	
Helper class for <code>setLogger</code>	70
<a href="#">CustomMeasurementBase</a>	
Helper class for custom measurements in Python and C#	71
<a href="#">DelayedChannel</a>	
Simple delayed queue	76
<a href="#">Experimental::DlsSignalGenerator</a>	82
<a href="#">Dump</a>	
<a href="#">Dump</a> all time tags to a file	86
<a href="#">Event</a>	
Object for the return value of <a href="#">Scope::getData</a>	89

<a href="#">EventGenerator</a>	
Generate predefined events in a virtual channel relative to a trigger event	90
<a href="#">Experimental::ExponentialSignalGenerator</a>	94
<a href="#">FastBinning</a>	
Helper class for fast division with a constant divisor	98
<a href="#">Experimental::FcsSignalGenerator</a>	100
<a href="#">FileReader</a>	
Reads tags from the disk files, which has been created by <a href="#">FileWriter</a>	104
<a href="#">FileWriter</a>	
Compresses and stores all time tags to a file	107
<a href="#">Flim</a>	
Fluorescence lifetime imaging	112
<a href="#">FlimAbstract</a>	
Interface for FLIM measurements, <a href="#">Flim</a> and <a href="#">FlimBase</a> classes inherit from it	124
<a href="#">FlimBase</a>	
Basic measurement, containing a minimal set of features for efficiency purposes	132
<a href="#">FlimFrameInfo</a>	
Object for storing the state of <a href="#">Flim::getCurrentFrameEx</a>	137
<a href="#">FrequencyCounter</a>	
Calculate the phase of multiple channels at equidistant sampling points	140
<a href="#">FrequencyCounterData</a>	144
<a href="#">FrequencyMultiplier</a>	
The signal of an input channel is scaled up to a higher frequency according to the multiplier passed as a parameter	147
<a href="#">FrequencyStability</a>	
Allan deviation (and related metrics) calculator	151
<a href="#">FrequencyStabilityData</a>	
Return data object for <a href="#">FrequencyStability::getData</a>	156
<a href="#">Experimental::GammaSignalGenerator</a>	159
<a href="#">GatedChannel</a>	
An input channel is gated by a gate channel	163
<a href="#">Experimental::GaussianSignalGenerator</a>	167
<a href="#">Histogram</a>	
Accumulate time differences into a histogram	170
<a href="#">Histogram2D</a>	
A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy	175
<a href="#">HistogramLogBins</a>	
Accumulate time differences into a histogram with logarithmic increasing bin sizes	180
<a href="#">HistogramLogBinsData</a>	
Helper object as return value for <a href="#">HistogramLogBins::getDataObject</a>	185
<a href="#">HistogramND</a>	
A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy	187
<a href="#">HistogramNDImpl&lt; T &gt;</a>	191
<a href="#">Iterator</a>	
Deprecated simple event queue	191
<a href="#">IteratorBase</a>	
Base class for all iterators	195
<a href="#">Experimental::MarkovProcessGenerator</a>	205
<a href="#">OrderedBarrier</a>	
Helper for implementing parallel measurements	209
<a href="#">OrderedPipeline</a>	
Helper for implementing parallel measurements	210
<a href="#">OrderedBarrier::OrderInstance</a>	
Internal object for serialization	210
<a href="#">Experimental::OscillatorSimulation</a>	212
<a href="#">Experimental::PatternSignalGenerator</a>	216



Experimental::PhotonGenerator	219
Experimental::PhotonNumber	
Photon number resolution	224
Experimental::PulsePerSecondData	
Helper object as return value for <a href="#">PulsePerSecondMonitor::getDataObject</a>	227
Experimental::PulsePerSecondMonitor	
Monitors the synchronicity of 1 pulse per second (PPS) signals	230
Sampler	
Triggered sampling measurement	234
Scope	
Scope measurement	238
Experimental::SignalGeneratorBase	243
Experimental::SimDetector	247
Experimental::SimLifetime	249
Experimental::SimSignalSplitter	252
SoftwareClockState	256
StartStop	
Simple start-stop measurement	257
SynchronizedMeasurements	
Start, stop and clear several measurements synchronized	261
SyntheticSingleTag	
Synthetic trigger timetag generator	265
Tag	
Single event on a channel	268
TimeDifferences	
Accumulates the time differences between clicks on two channels in one or more histograms	271
TimeDifferencesImpl< T >	276
TimeDifferencesND	
Accumulates the time differences between clicks on two channels in a multi-dimensional histogram	277
TimeTagger	
Backend for the <a href="#">TimeTagger</a>	282
TimeTaggerBase	
Basis interface for all Time Tagger classes	304
TimeTaggerNetwork	
Network <a href="#">TimeTagger</a> client	316
TimeTaggerVirtual	
Virtual <a href="#">TimeTagger</a> based on dump files	330
TimeTagStream	
Access the time tag stream	336
TimeTagStreamBuffer	
Return object for <a href="#">TimeTagStream::getData</a>	340
Experimental::TransformCrosstalk	342
Experimental::TransformDeadtime	345
Experimental::TransformEfficiency	349
Experimental::TransformGaussianBroadening	353
TriggerOnCountrate	
Inject trigger events when exceeding or falling below a given count rate within a rolling time window	356
Experimental::TwoStateExponentialSignalGenerator	362
Experimental::UniformSignalGenerator	366



## Chapter 6

# File Index

### 6.1 File List

Here is a list of all files with brief descriptions:

<a href="#">Iterators.h</a> . . . . .	371
<a href="#">TimeTagger.h</a> . . . . .	398



## Chapter 7

# Topic Documentation

### 7.1 Implementations with a Time Tagger interface

#### Classes

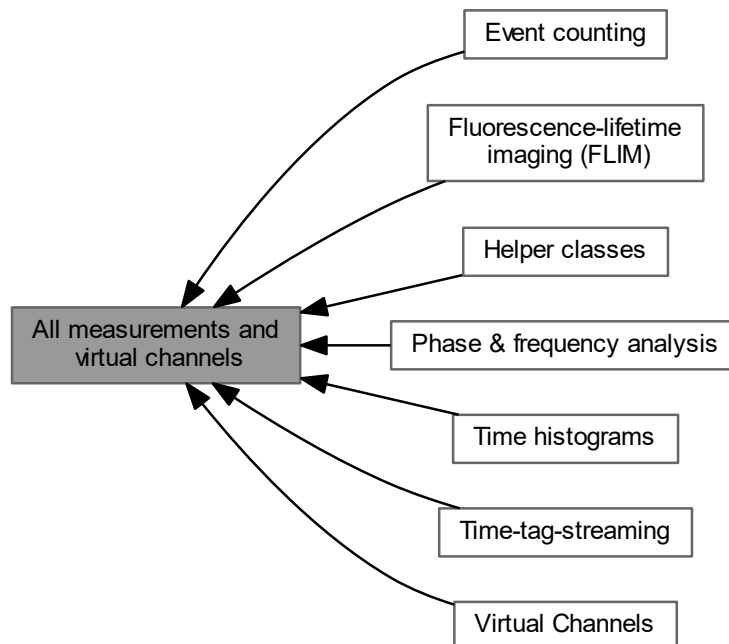
- class [TimeTaggerBase](#)  
*Basis interface for all Time Tagger classes.*
- class [TimeTaggerVirtual](#)  
*virtual [TimeTagger](#) based on dump files*
- class [TimeTagger](#)  
*backend for the [TimeTagger](#).*

#### 7.1.1 Detailed Description

### 7.2 All measurements and virtual channels

Base iterators for photon counting applications.

Collaboration diagram for All measurements and virtual channels:



## Topics

- [Event counting](#)
- [Time histograms](#)

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

- [Fluorescence-lifetime imaging \(FLIM\)](#)

*This section describes the [Flim](#) related measurements classes of the Time Tagger API.*

- [Phase & frequency analysis](#)

*This section describes measurement classes that expect periodic signals.*

- [Time-tag-streaming](#)

*Measurement classes described in this section provide direct access to the time tag stream with minimal or no pre-processing.*

- [Helper classes](#)
- [Virtual Channels](#)

## Classes

- class [HistogramND](#)

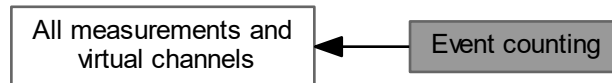
*A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.*

## 7.2.1 Detailed Description

Base iterators for photon counting applications.

### 7.2.2 Event counting

Collaboration diagram for Event counting:



#### Classes

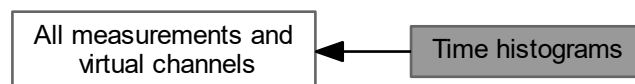
- class [CountBetweenMarkers](#)  
*a simple counter where external marker signals determine the bins*
- class [Counter](#)  
*a simple counter on one or more channels*
- class [Countrate](#)  
*count rate on one or more channels*

#### 7.2.2.1 Detailed Description

### 7.2.3 Time histograms

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

Collaboration diagram for Time histograms:



## Classes

- class [StartStop](#)  
*simple start-stop measurement*
- class [TimeDifferences](#)  
*Accumulates the time differences between clicks on two channels in one or more histograms.*
- class [Histogram2D](#)  
*A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.*
- class [TimeDifferencesND](#)  
*Accumulates the time differences between clicks on two channels in a multi-dimensional histogram.*
- class [Histogram](#)  
*Accumulate time differences into a histogram.*
- class [HistogramLogBins](#)  
*Accumulate time differences into a histogram with logarithmic increasing bin sizes.*
- class [Correlation](#)  
*Auto- and Cross-correlation measurement.*

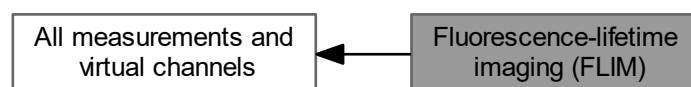
### 7.2.3.1 Detailed Description

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

## 7.2.4 Fluorescence-lifetime imaging (FLIM)

This section describes the [Flim](#) related measurements classes of the Time Tagger API.

Collaboration diagram for Fluorescence-lifetime imaging (FLIM):



## Classes

- class [FlimBase](#)  
*basic measurement, containing a minimal set of features for efficiency purposes*
- class [Flim](#)  
*Fluorescence lifetime imaging.*

### 7.2.4.1 Detailed Description

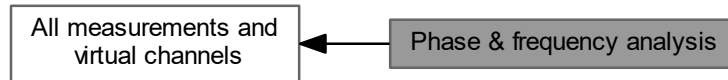
This section describes the [Flim](#) related measurements classes of the Time Tagger API.



### 7.2.5 Phase & frequency analysis

This section describes measurement classes that expect periodic signals.

Collaboration diagram for Phase & frequency analysis:



#### Classes

- class [FrequencyCounter](#)  
*Calculate the phase of multiple channels at equidistant sampling points.*
- class [FrequencyStability](#)  
*Allan deviation (and related metrics) calculator.*
- class [Experimental::PulsePerSecondMonitor](#)  
*Monitors the synchronicity of 1 pulse per second (PPS) signals.*

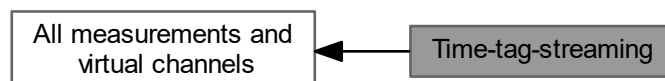
#### 7.2.5.1 Detailed Description

This section describes measurement classes that expect periodic signals.

### 7.2.6 Time-tag-streaming

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

Collaboration diagram for Time-tag-streaming:



## Classes

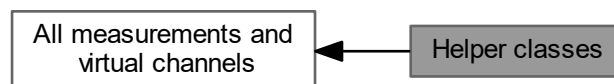
- class [Iterator](#)  
*a deprecated simple event queue*
- class [TimeTagStream](#)  
*access the time tag stream*
- class [Dump](#)  
*dump all time tags to a file*
- class [Scope](#)  
*a scope measurement*
- class [FileWriter](#)  
*compresses and stores all time tags to a file*
- class [FileReader](#)  
*Reads tags from the disk files, which has been created by [FileWriter](#).*
- class [Sampler](#)  
*a triggered sampling measurement*

### 7.2.6.1 Detailed Description

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

## 7.2.7 Helper classes

Collaboration diagram for Helper classes:



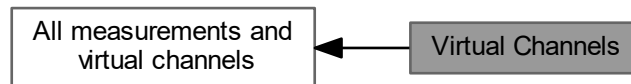
## Classes

- class [SynchronizedMeasurements](#)  
*start, stop and clear several measurements synchronized*
- class [CustomMeasurementBase](#)  
*Helper class for custom measurements in Python and C#.*
- class [SyntheticSingleTag](#)  
*synthetic trigger timetag generator.*

### 7.2.7.1 Detailed Description

## 7.2.8 Virtual Channels

Collaboration diagram for Virtual Channels:



### Classes

- class [Combiner](#)  
*Combine some channels in a virtual channel which has a tick for each tick in the input channels.*
- class [Coincidences](#)  
*a coincidence monitor for many channel groups*
- class [Coincidence](#)  
*a coincidence monitor for one channel group*
- class [DelayedChannel](#)  
*a simple delayed queue*
- class [TriggerOnCountRate](#)  
*Inject trigger events when exceeding or falling below a given count rate within a rolling time window.*
- class [GatedChannel](#)  
*An input channel is gated by a gate channel.*
- class [FrequencyMultiplier](#)  
*The signal of an input channel is scaled up to a higher frequency according to the multiplier passed as a parameter.*
- class [ConstantFractionDiscriminator](#)  
*a virtual CFD implementation which returns the mean time between a rising and a falling pair of edges*
- class [EventGenerator](#)  
*Generate predefined events in a virtual channel relative to a trigger event.*
- class [Combinations](#)
- class [Experimental::PhotonNumber](#)  
*Photon number resolution.*

### 7.2.8.1 Detailed Description

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.



## Chapter 8

# Namespace Documentation

### 8.1 Experimental Namespace Reference

Namespace for features, which are still in development and are likely to change.

#### Classes

- class [DlsSignalGenerator](#)
- class [ExponentialSignalGenerator](#)
- class [FcsSignalGenerator](#)
- class [GammaSignalGenerator](#)
- class [GaussianSignalGenerator](#)
- class [MarkovProcessGenerator](#)
- class [OscillatorSimulation](#)
- class [PatternSignalGenerator](#)
- class [PhotonGenerator](#)
- class [PhotonNumber](#)  
*Photon number resolution.*
- class [PulsePerSecondData](#)  
*Helper object as return value for [PulsePerSecondMonitor::getDataObject](#).*
- class [PulsePerSecondMonitor](#)  
*Monitors the synchronicity of 1 pulse per second (PPS) signals.*
- class [SignalGeneratorBase](#)
- class [SimDetector](#)
- class [SimLifetime](#)
- class [SimSignalSplitter](#)
- class [TransformCrosstalk](#)
- class [TransformDeadtime](#)
- class [TransformEfficiency](#)
- class [TransformGaussianBroadening](#)
- class [TwoStateExponentialSignalGenerator](#)
- class [UniformSignalGenerator](#)

#### 8.1.1 Detailed Description

Namespace for features, which are still in development and are likely to change.



## Chapter 9

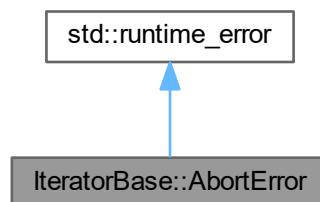
# Class Documentation

### 9.1 IteratorBase::AbortError Class Reference

A custom runtime error thrown by the `abort` call. This can be caught and handled by measurement classes, including `CustomMeasurement`, to perform actions within the abortion process.

```
#include <TimeTagger.h>
```

Inheritance diagram for `IteratorBase::AbortError`:



#### Public Member Functions

- `AbortError` (const std::string &what\_arg)
- `~AbortError` ()

#### 9.1.1 Detailed Description

A custom runtime error thrown by the `abort` call. This can be caught and handled by measurement classes, including `CustomMeasurement`, to perform actions within the abortion process.

## 9.1.2 Constructor & Destructor Documentation

### 9.1.2.1 `AbortError()`

```
IteratorBase::AbortError::AbortError (
    const std::string & what_arg ) [inline]
```

### 9.1.2.2 `~AbortError()`

```
IteratorBase::AbortError::~~AbortError ( ) [inline]
```

The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.2 ChannelGate Struct Reference

```
#include <Iterators.h>
```

### Public Member Functions

- [ChannelGate](#) ([channel\\_t](#) gate\_open\_channel, [channel\\_t](#) gate\_close\_channel, [GatedChannelInitial](#) initial=[GatedChannelInitial::Open](#))

### Public Attributes

- const [channel\\_t](#) gate\_open\_channel
- const [channel\\_t](#) gate\_close\_channel
- const [GatedChannelInitial](#) initial

## 9.2.1 Constructor & Destructor Documentation

### 9.2.1.1 `ChannelGate()`

```
ChannelGate::ChannelGate (
    channel\_t gate_open_channel,
    channel\_t gate_close_channel,
    GatedChannelInitial initial = GatedChannelInitial::Open ) [inline]
```

## 9.2.2 Member Data Documentation

### 9.2.2.1 `gate_close_channel`

```
const channel\_t ChannelGate::gate_close_channel
```



### 9.2.2.2 gate\_open\_channel

```
const channel_t ChannelGate::gate_open_channel
```

### 9.2.2.3 initial

```
const GatedChannelInitial ChannelGate::initial
```

The documentation for this struct was generated from the following file:

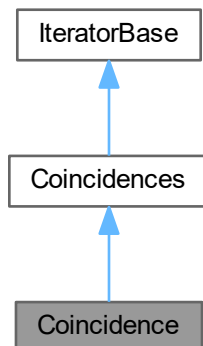
- [Iterators.h](#)

## 9.3 Coincidence Class Reference

a coincidence monitor for one channel group

```
#include <Iterators.h>
```

Inheritance diagram for Coincidence:



### Public Member Functions

- [Coincidence](#) ([TimeTaggerBase](#) \*tagger, std::vector< [channel\\_t](#) > channels, [timestamp\\_t](#) coincidence←Window=1000, [CoincidenceTimestamp](#) timestamp=[CoincidenceTimestamp::Last](#))  
*construct a coincidence*
- [channel\\_t](#) [getChannel](#) ()  
*virtual channel which contains the coincidences*

## Public Member Functions inherited from **Coincidences**

- **Coincidences** (**TimeTaggerBase** \*tagger, std::vector< std::vector< **channel\_t** > > coincidenceGroups, **timestamp\_t** coincidenceWindow, **CoincidenceTimestamp** timestamp=**CoincidenceTimestamp::Last**)  
*construct a **Coincidences***
- **~Coincidences** ()
- std::vector< **channel\_t** > **getChannels** ()  
*fetches the block of virtual channels for those coincidence groups*
- void **setCoincidenceWindow** (**timestamp\_t** coincidenceWindow)

## Public Member Functions inherited from **IteratorBase**

- virtual **~IteratorBase** ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void **start** ()  
*Starts or continues data acquisition.*
- void **startFor** (**timestamp\_t** capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool **waitUntilFinished** (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with **startFor()**.*
- void **stop** ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void **clear** ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void **abort** ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool **isRunning** ()  
*Returns True if the measurement is collecting the data.*
- **timestamp\_t** **getCaptureDuration** ()  
*Total capture duration since the measurement creation or last call to **clear()**.*
- std::string **getConfiguration** ()  
*Fetches the overall configuration status of the measurement.*

## Additional Inherited Members

## Protected Member Functions inherited from **Coincidences**

- bool **next\_impl** (std::vector< **Tag** > &incoming\_tags, **timestamp\_t** begin\_time, **timestamp\_t** end\_time) override  
*update iterator state*

## Protected Member Functions inherited from [IteratorBase](#)

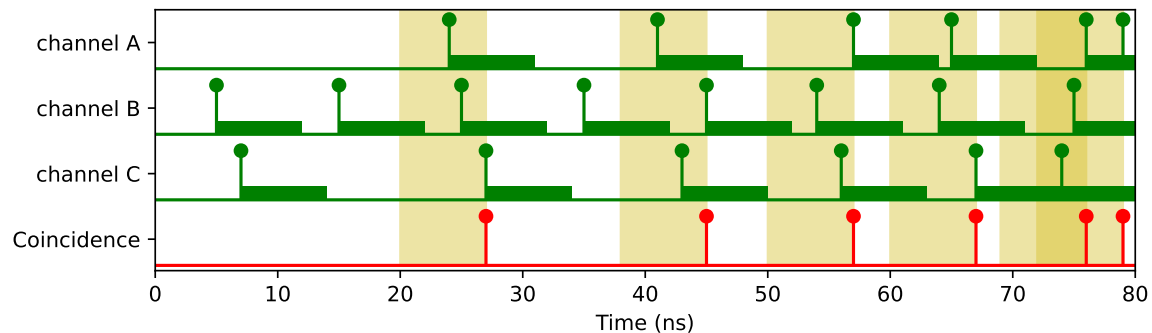
- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.3.1 Detailed Description

a coincidence monitor for one channel group



Monitor coincidences for a given channel groups passed by the constructor. A coincidence is event is detected when all selected channels have a click within the given coincidenceWindow [ps] The coincidence will create a virtual events on a virtual channel with the channel number provided by [getChannel\(\)](#). For multiple coincidence channel combinations use the class [Coincidences](#) which outperforms multiple instances of [Coincidence](#).

### 9.3.2 Constructor & Destructor Documentation

#### 9.3.2.1 Coincidence()

```
Coincidence::Coincidence (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels,
    timestamp_t coincidenceWindow = 1000,
    CoincidenceTimestamp timestamp = CoincidenceTimestamp::Last ) [inline]
```

construct a coincidence

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>channels</i>	vector of channels to match
<i>coincidenceWindow</i>	max distance between all clicks for a coincidence [ps]
<i>timestamp</i>	type of timestamp for virtual channel (Last, Average, First, ListedFirst)

### 9.3.3 Member Function Documentation

#### 9.3.3.1 getChannel()

```
channel_t Coincidence::getChannel ( ) [inline]
```

virtual channel which contains the coincidences

The documentation for this class was generated from the following file:

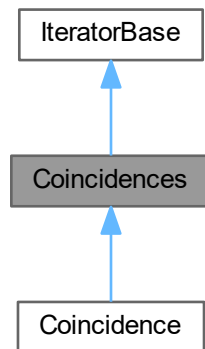
- [Iterators.h](#)

## 9.4 Coincidences Class Reference

a coincidence monitor for many channel groups

```
#include <Iterators.h>
```

Inheritance diagram for Coincidences:



### Public Member Functions

- **Coincidences** (**TimeTaggerBase** \*tagger, std::vector< std::vector< **channel\_t** > > coincidenceGroups, **timestamp\_t** coincidenceWindow, **CoincidenceTimestamp** timestamp=**CoincidenceTimestamp::Last**)  
construct a *Coincidences*
- **~Coincidences** ()
- std::vector< **channel\_t** > **getChannels** ()  
fetches the block of virtual channels for those coincidence groups
- void **setCoincidenceWindow** (**timestamp\_t** coincidenceWindow)

### Public Member Functions inherited from **IteratorBase**

- virtual **~IteratorBase** ()  
destructor, will unregister from the Time Tagger prior finalization.
- void **start** ()  
Starts or continues data acquisition.
- void **startFor** (**timestamp\_t** capture\_duration, bool clear=true)  
Starts or continues the data acquisition for the given duration.
- bool **waitUntilFinished** (int64\_t timeout=-1)  
Blocks the execution until the measurement has finished. Can be used with **startFor()**.
- void **stop** ()  
After calling this method, the measurement will stop processing incoming tags.
- void **clear** ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.

- void `abort ()`  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning ()`  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration ()`  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration ()`  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl (std::vector< Tag > &incoming_tags, timestamp_t begin_time, timestamp_t end_time)` override  
*update iterator state*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase (TimeTaggerBase *tagger, std::string base_type_="IteratorBase", std::string extra_info_="")`  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel (channel_t channel)`  
*register a channel*
- void `unregisterChannel (channel_t channel)`  
*unregister a channel*
- `channel_t getNewVirtualChannel ()`  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization ()`  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl ()`  
*clear `Iterator` state.*
- virtual void `on_start ()`  
*callback when the measurement class is started*
- virtual void `on_stop ()`  
*callback when the measurement class is stopped*
- void `lock ()`  
*acquire update lock*
- void `unlock ()`  
*release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`  
*acquire update lock*
- void `finish_running ()`  
*Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`  
void `checkForAbort (T callback)`

## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.4.1 Detailed Description

a coincidence monitor for many channel groups

Monitor coincidences for given coincidence groups passed by the constructor. A coincidence is hereby defined as for a given coincidence group a) the incoming is part of this group b) at least tag arrived within the coincidence↔ Window [ps] for all other channels of this coincidence group Each coincidence will create a virtual event. The block of event IDs for those coincidence group can be fetched.

### 9.4.2 Constructor & Destructor Documentation

#### 9.4.2.1 Coincidences()

```
Coincidences::Coincidences (
    TimeTaggerBase * tagger,
    std::vector< std::vector< channel\_t > > coincidenceGroups,
    timestamp\_t coincidenceWindow,
    CoincidenceTimestamp timestamp = CoincidenceTimestamp::Last )
```

construct a [Coincidences](#)

#### Parameters

<i><a href="#">tagger</a></i>	reference to a <a href="#">TimeTagger</a>
<i><a href="#">coincidenceGroups</a></i>	a vector of channels defining the coincidences
<i><a href="#">coincidenceWindow</a></i>	the size of the coincidence window in picoseconds
<i><a href="#">timestamp</a></i>	type of timestamp for virtual channel (Last, Average, First, ListedFirst)

### 9.4.2.2 ~Coincidences()

```
Coincidences::~~Coincidences ( )
```

## 9.4.3 Member Function Documentation

### 9.4.3.1 getChannels()

```
std::vector< channel_t > Coincidences::getChannels ( )
```

fetches the block of virtual channels for those coincidence groups

### 9.4.3.2 next\_impl()

```
bool Coincidences::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.4.3.3 setCoincidenceWindow()

```
void Coincidences::setCoincidenceWindow (
    timestamp_t coincidenceWindow )
```

The documentation for this class was generated from the following file:

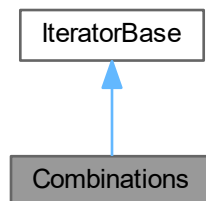
- [Iterators.h](#)



## 9.5 Combinations Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Combinations:



### Public Member Functions

- `Combinations` (`TimeTaggerBase` \*`tagger`, `std::vector`< `channel_t` > `const` &`channels`, `timestamp_t` `window`↔  
\_size)  
construct a `Combinations`
- `~Combinations` ()
- `channel_t` `getChannel` (`std::vector`< `channel_t` > `const` &`input_channels`) `const`  
Return the virtual channel ID corresponding to an exclusive coincidence on the given `input_channels`. The channel gets implicitly enabled.
- `channel_t` `getSumChannel` (`int` `n_channels`) `const`  
return the ID of the virtual channel corresponding to an `n_channel`-fold combination of input channels
- `std::vector`< `channel_t` > `getCombination` (`channel_t` `virtual_channel`) `const`  
Return the set of input channels that emit a coincidence event on the given virtual channel `virtual_channel`.

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
destructor, will unregister from the Time Tagger prior finalization.
- void `start` ()  
Starts or continues data acquisition.
- void `startFor` (`timestamp_t` `capture_duration`, bool `clear`=true)  
Starts or continues the data acquisition for the given duration.
- bool `waitUntilFinished` (`int64_t` `timeout`=-1)  
Blocks the execution until the measurement has finished. Can be used with `startFor()`.
- void `stop` ()  
After calling this method, the measurement will stop processing incoming tags.
- void `clear` ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- void `abort` ()

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

- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (`std::vector`< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock`< `std::mutex` > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- `template`<typename T >  
void `checkForAbort` (T callback)

## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the `finishInitialization` callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.5.1 Detailed Description

A combination is defined as one or more events occurring on distinct channels within a given time window, preceded and followed by two guard windows of the same duration without any events on these channels. The guard window starts at the time of the last event during the coincidence window.

This iterator emits events on virtual channels whenever such a combination is detected on the monitored channels. Given  $N$  input channels  $c_1, \dots, c_N$ , there are  $2^N - 1$  possible combination, each having a corresponding virtual channel.

The individual virtual channels one is interested in have to be enabled by calling [getChannel\(\)](#) before clicks on them are actually generated.

Additionally, there are  $N$  "sum channels". The  $n$ -th sum channel generates a clicks on each  $n$ -fold combination (irrespective of the specific contributing input channel).

Note that multiple events on the same channel during the coincidence window are counted as one.

### 9.5.2 Constructor & Destructor Documentation

#### 9.5.2.1 Combinations()

```
Combinations::Combinations (
    TimeTaggerBase * tagger,
    std::vector< channel\_t > const & channels,
    timestamp\_t window\_size )
```

construct a [Combinations](#)

#### Parameters

<i><a href="#">tagger</a></i>	reference to a <a href="#">TimeTagger</a>
<i><a href="#">channels</a></i>	the set of channels to monitor. Elements must be distinct.
<i><a href="#">window_size</a></i>	duration of the coincidence window

### 9.5.2.2 ~Combinations()

```
Combinations::~~Combinations ( )
```

## 9.5.3 Member Function Documentation

### 9.5.3.1 clear\_impl()

```
void Combinations::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.5.3.2 getChannel()

```
channel_t Combinations::getChannel (
    std::vector< channel_t > const & input_channels ) const
```

Return the virtual channel ID corresponding to an exclusive coincidence on the given *input\_channels*. The channel gets implicitly enabled.

### 9.5.3.3 getCombination()

```
std::vector< channel_t > Combinations::getCombination (
    channel_t virtual_channel ) const
```

Return the set of input channels that emit a coincidence event on the given virtual channel *virtual\_channel*.

### 9.5.3.4 getSumChannel()

```
channel_t Combinations::getSumChannel (
    int n_channels ) const
```

return the ID of the virtual channel corresponding to an *n\_channel*-fold combination of input channels

### 9.5.3.5 next\_impl()

```
bool Combinations::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

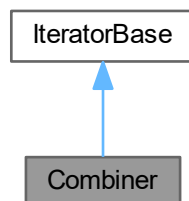
- [Iterators.h](#)

## 9.6 Combiner Class Reference

Combine some channels in a virtual channel which has a tick for each tick in the input channels.

```
#include <Iterators.h>
```

Inheritance diagram for Combiner:



## Public Member Functions

- [Combiner](#) ([TimeTaggerBase](#) \*tagger, std::vector< [channel\\_t](#) > channels)  
*construct a combiner*
- [~Combiner](#) ()
- void [getChannelCounts](#) (std::function< int64\_t \*(size\_t)> array\_out)  
*get sum of counts*
- void [getData](#) (std::function< int64\_t \*(size\_t)> array\_out)  
*get sum of counts*
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- template<typename T >  
void `checkForAbort (T callback)`

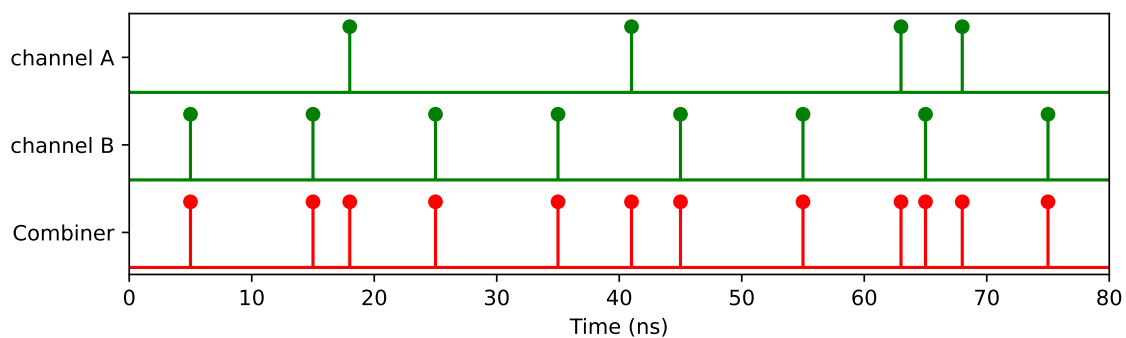
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.6.1 Detailed Description

Combine some channels in a virtual channel which has a tick for each tick in the input channels.



This iterator can be used to get aggregation channels, eg if you want to monitor the countrate of the sum of two channels.

## 9.6.2 Constructor & Destructor Documentation

### 9.6.2.1 Combiner()

```
Combiner::Combiner (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels )
```

construct a combiner

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>channels</i>	vector of channels to combine

### 9.6.2.2 ~Combiner()

```
Combiner::~Combiner ( )
```

## 9.6.3 Member Function Documentation

### 9.6.3.1 clear\_impl()

```
void Combiner::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.6.3.2 getChannel()

```
channel_t Combiner::getChannel ( )
```

the new virtual channel

This function returns the new allocated virtual channel. It can be used now in any new iterator.

### 9.6.3.3 getChannelCounts()

```
void Combiner::getChannelCounts (
    std::function< int64_t *(size_t)> array_out )
```

get sum of counts

For reference, this iterators sums up how much ticks are generated because of which input channel. So this functions returns an array with one value per input channel.



#### 9.6.3.4 getData()

```
void Combiner::getData (
    std::function< int64_t *(size_t)> array_out )
```

get sum of counts

deprecated, use getChannelCounts instead.

#### 9.6.3.5 next\_impl()

```
bool Combiner::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

##### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

##### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

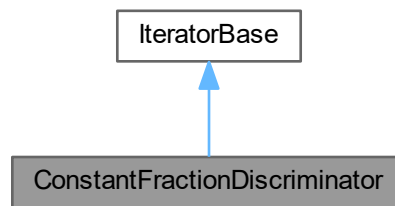
- [Iterators.h](#)

## 9.7 ConstantFractionDiscriminator Class Reference

a virtual CFD implementation which returns the mean time between a rising and a falling pair of edges

```
#include <Iterators.h>
```

Inheritance diagram for ConstantFractionDiscriminator:



### Public Member Functions

- [ConstantFractionDiscriminator](#) ([TimeTaggerBase](#) \*tagger, [std::vector](#)< [channel\\_t](#) > channels, [timestamp\\_t](#) search\_window)  
*constructor of a [ConstantFractionDiscriminator](#)*
- [~ConstantFractionDiscriminator](#) ()
- [std::vector](#)< [channel\\_t](#) > [getChannels](#) ()  
*the list of new virtual channels*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- [std::string](#) [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

**Protected Member Functions**

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `on_start` () override  
*callback when the measurement class is started*

**Protected Member Functions inherited from `IteratorBase`**

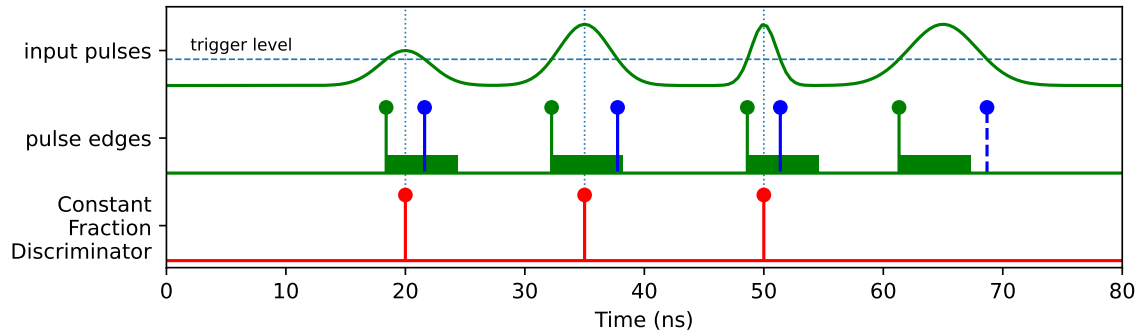
- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

**Additional Inherited Members****Protected Attributes inherited from `IteratorBase`**

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

## 9.7.1 Detailed Description

a virtual CFD implementation which returns the mean time between a rising and a falling pair of edges



## 9.7.2 Constructor & Destructor Documentation

### 9.7.2.1 ConstantFractionDiscriminator()

```
ConstantFractionDiscriminator::ConstantFractionDiscriminator (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels,
    timestamp_t search_window )
```

constructor of a [ConstantFractionDiscriminator](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>channels</i>	list of channels for the CFD, the formers of the rising+falling pairs must be given
<i>search_window</i>	interval for the CFD window, must be positive

### 9.7.2.2 ~ConstantFractionDiscriminator()

```
ConstantFractionDiscriminator::~~ConstantFractionDiscriminator ( )
```

## 9.7.3 Member Function Documentation

### 9.7.3.1 getChannels()

```
std::vector< channel_t > ConstantFractionDiscriminator::getChannels ( )
```

the list of new virtual channels

This function returns the list of new allocated virtual channels. It can be used now in any new measurement class.

### 9.7.3.2 next\_impl()

```
bool ConstantFractionDiscriminator::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.7.3.3 on\_start()

```
void ConstantFractionDiscriminator::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

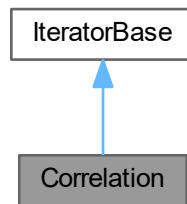
- [Iterators.h](#)

## 9.8 Correlation Class Reference

Auto- and Cross-correlation measurement.

```
#include <Iterators.h>
```

Inheritance diagram for Correlation:



### Public Member Functions

- `Correlation (TimeTaggerBase *tagger, channel_t channel_1, channel_t channel_2=CHANNEL_UNUSED, timestamp_t binwidth=1000, int n_bins=1000)`  
*constructor of a correlation measurement*
- `~Correlation ()`  
*destructor of the `Correlation` measurement*
- `void getData (std::function< int32_t *(size_t)> array_out)`  
*returns a one-dimensional array of size `n_bins` containing the histogram*
- `void getDataNormalized (std::function< double *(size_t)> array_out)`  
*get the  $g(2)$  normalized histogram*
- `void getIndex (std::function< timestamp_t *(size_t)> array_out)`  
*returns a vector of size `n_bins` containing the time bins in ps*

### Public Member Functions inherited from `IteratorBase`

- `virtual ~IteratorBase ()`  
*destructor, will unregister from the Time Tagger prior finalization.*
- `void start ()`  
*Starts or continues data acquisition.*
- `void startFor (timestamp_t capture_duration, bool clear=true)`  
*Starts or continues the data acquisition for the given duration.*
- `bool waitUntilFinished (int64_t timeout=-1)`  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- `void stop ()`  
*After calling this method, the measurement will stop processing incoming tags.*
- `void clear ()`  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- `void abort ()`  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- `bool isRunning ()`  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration ()`  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration ()`  
*Fetches the overall configuration status of the measurement.*

**Protected Member Functions**

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*

**Protected Member Functions inherited from `IteratorBase`**

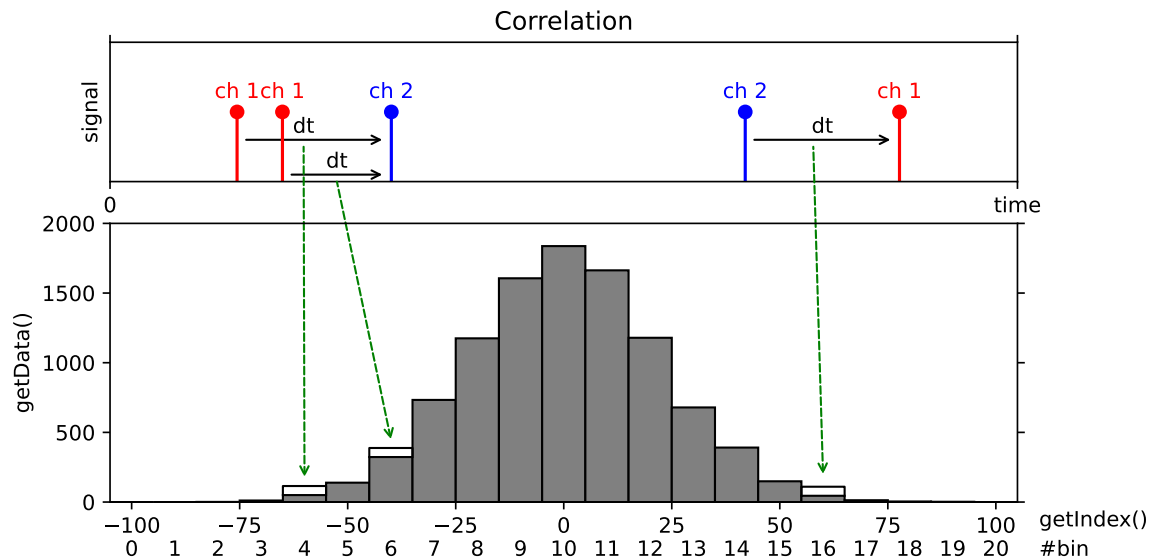
- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

**Additional Inherited Members****Protected Attributes inherited from `IteratorBase`**

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

## 9.8.1 Detailed Description

Auto- and Cross-correlation measurement.



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.

## 9.8.2 Constructor & Destructor Documentation

### 9.8.2.1 Correlation()

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

constructor of a correlation measurement

#### Note

When channel\_1 is left empty or set to CHANNEL\_UNUSED -> an auto-correlation measurement is performed, which is the same as setting channel\_1 = channel\_2.

#### Parameters

<i>tagger</i>	time tagger object
<i>channel_1</i>	channel on which (stop) clicks are received
<i>channel_2</i>	channel on which reference clicks (start) are received
<i>binwidth</i>	bin width in ps
<i>n_bins</i>	the number of bins in the resulting histogram



### 9.8.2.2 ~Correlation()

```
Correlation::~~Correlation ( )
```

destructor of the [Correlation](#) measurement

## 9.8.3 Member Function Documentation

### 9.8.3.1 clear\_impl()

```
void Correlation::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.8.3.2 getData()

```
void Correlation::getData (
    std::function< int32_t *(size_t)> array_out )
```

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

#### Parameters

<code>array_out</code>	allocator callback for managed return values
------------------------	--

### 9.8.3.3 getDataNormalized()

```
void Correlation::getDataNormalized (
    std::function< double *(size_t)> array_out )
```

get the  $g(2)$  normalized histogram

Return the data normalized as:  $g^{(2)}(\tau) = \frac{\Delta t}{\text{binwidth}(\tau) \cdot N_1 \cdot N_2} \cdot \text{histogram}(\tau)$

This is normalized in such a way that a perfectly uncorrelated signals would result in a histogram with a mean value of bins equal to one.

#### Parameters

<code>array_out</code>	allocator callback for managed return values
------------------------	--

#### 9.8.3.4 getIndex()

```
void Correlation::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

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

##### Parameters

<code>array_out</code>	allocator callback for managed return values
------------------------	--

#### 9.8.3.5 next\_impl()

```
bool Correlation::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

##### Parameters

<code>incoming_tags</code>	block of events
<code>begin_time</code>	earliest event in the block
<code>end_time</code>	begin_time of the next block, not including in this block

##### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

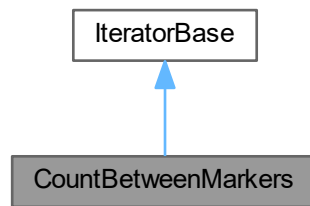
- [Iterators.h](#)

## 9.9 CountBetweenMarkers Class Reference

a simple counter where external marker signals determine the bins

```
#include <Iterators.h>
```

Inheritance diagram for CountBetweenMarkers:



### Public Member Functions

- `CountBetweenMarkers` (`TimeTaggerBase` \*tagger, `channel_t` click\_channel, `channel_t` begin\_channel, `channel_t` end\_channel=`CHANNEL_UNUSED`, `int32_t` n\_values=1000)  
*constructor of `CountBetweenMarkers`*
- `~CountBetweenMarkers` ()
- `bool` `ready` ()  
*Returns true when the entire array is filled.*
- `void` `getData` (`std::function`< `int32_t` \*(`size_t`)> array\_out)  
*Returns array of size n\_values containing the acquired counter values.*
- `void` `getBinWidths` (`std::function`< `timestamp_t` \*(`size_t`)> array\_out)  
*fetches the widths of each bins*
- `void` `getIndex` (`std::function`< `timestamp_t` \*(`size_t`)> array\_out)  
*fetches the starting time of each bin*

### Public Member Functions inherited from `IteratorBase`

- `virtual` `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- `void` `start` ()  
*Starts or continues data acquisition.*
- `void` `startFor` (`timestamp_t` capture\_duration, `bool` clear=true)  
*Starts or continues the data acquisition for the given duration.*
- `bool` `waitUntilFinished` (`int64_t` timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- `void` `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- `void` `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- `void` `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- `bool` `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

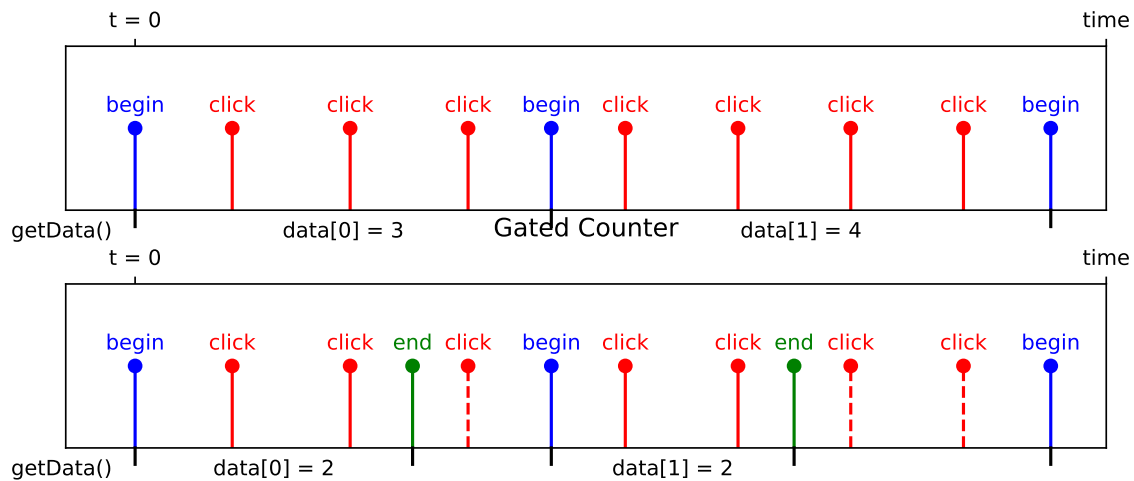
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

### 9.9.1 Detailed Description

a simple counter where external marker signals determine the bins



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

### 9.9.2 Constructor & Destructor Documentation

#### 9.9.2.1 CountBetweenMarkers()

```
CountBetweenMarkers::CountBetweenMarkers (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t begin_channel,
    channel_t end_channel = CHANNEL_UNUSED,
    int32_t n_values = 1000 )
```

constructor of [CountBetweenMarkers](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel that increases the count
<i>begin_channel</i>	channel that triggers beginning of counting and stepping to the next value
<i>end_channel</i>	channel that triggers end of counting
<i>n_values</i>	the number of counter values to be stored

### 9.9.2.2 ~CountBetweenMarkers()

```
CountBetweenMarkers::~~CountBetweenMarkers ( )
```

## 9.9.3 Member Function Documentation

### 9.9.3.1 clear\_impl()

```
void CountBetweenMarkers::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.9.3.2 getBinWidths()

```
void CountBetweenMarkers::getBinWidths (
    std::function< timestamp\_t *(size_t)> array_out )
```

fetches the widths of each bins

### 9.9.3.3 getData()

```
void CountBetweenMarkers::getData (
    std::function< int32\_t *(size_t)> array_out )
```

Returns array of size `n_values` containing the acquired counter values.

### 9.9.3.4 getIndex()

```
void CountBetweenMarkers::getIndex (
    std::function< timestamp\_t *(size_t)> array_out )
```

fetches the starting time of each bin

### 9.9.3.5 next\_impl()

```
bool CountBetweenMarkers::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp\_t begin_time,
    timestamp\_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.9.3.6 ready()

```
bool CountBetweenMarkers::ready ( )
```

Returns true when the entire array is filled.

The documentation for this class was generated from the following file:

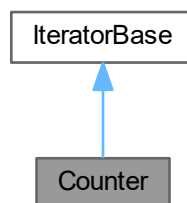
- [Iterators.h](#)

## 9.10 Counter Class Reference

a simple counter on one or more channels

```
#include <Iterators.h>
```

Inheritance diagram for Counter:



## Public Member Functions

- **Counter** (**TimeTaggerBase** \*tagger, std::vector< **channel\_t** > channels, **timestamp\_t** binwidth=1000000000, int32\_t n\_values=1)  
*construct a counter*
- **~Counter** ()
- void **getData** (std::function< int32\_t \*(size\_t, size\_t)> array\_out, bool rolling=true)  
*An array of size 'number of channels' by n\_values containing the current values of the circular buffer (counts in each bin).*
- void **getDataNormalized** (std::function< double \*(size\_t, size\_t)> array\_out, bool rolling=true)  
*get countrate in Hz*
- void **getDataTotalCounts** (std::function< uint64\_t \*(size\_t)> array\_out)  
*get the total amount of clicks per channel since the last clear including the currently integrating bin*
- void **getIndex** (std::function< **timestamp\_t** \*(size\_t)> array\_out)  
*A vector of size n\_values containing the time bins in ps.*
- **CounterData** **getDataObject** (bool remove=false)  
*Fetch the most recent up to n\_values bins, which have not been removed before.*

## Public Member Functions inherited from **IteratorBase**

- virtual **~IteratorBase** ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void **start** ()  
*Starts or continues data acquisition.*
- void **startFor** (**timestamp\_t** capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool **waitUntilFinished** (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with **startFor()**.*
- void **stop** ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void **clear** ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void **abort** ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool **isRunning** ()  
*Returns True if the measurement is collecting the data.*
- **timestamp\_t** **getCaptureDuration** ()  
*Total capture duration since the measurement creation or last call to **clear()**.*
- std::string **getConfiguration** ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool **next\_impl** (std::vector< **Tag** > &incoming\_tags, **timestamp\_t** begin\_time, **timestamp\_t** end\_time) override  
*update iterator state*
- void **clear\_impl** () override  
*clear **Iterator** state.*
- void **on\_start** () override  
*callback when the measurement class is started*



## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

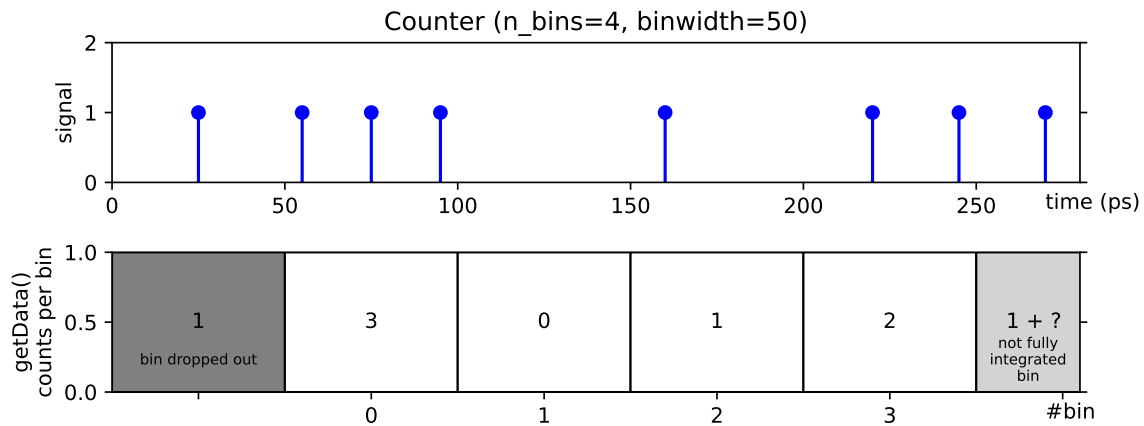
## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.10.1 Detailed Description

a simple counter on one or more channels



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

### 9.10.2 Constructor & Destructor Documentation

#### 9.10.2.1 Counter()

```
Counter::Counter (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels,
    timestamp_t binwidth = 1000000000,
    int32_t n_values = 1 )
```

construct a counter

##### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>channels</i>	channels to count on
<i>binwidth</i>	counts are accumulated for binwidth picoseconds
<i>n_values</i>	number of counter values stored (for each channel)

#### 9.10.2.2 ~Counter()

```
Counter::~Counter ( )
```

### 9.10.3 Member Function Documentation

#### 9.10.3.1 clear\_impl()

```
void Counter::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.10.3.2 getData()

```
void Counter::getData (
    std::function< int32_t *(size_t, size_t)> array_out,
    bool rolling = true )
```

An array of size 'number of channels' by n\_values containing the current values of the circular buffer (counts in each bin).

##### Parameters

<i>array_out</i>	allocator callback for managed return values
<i>rolling</i>	if true, the returning array starts with the oldest data and goes up to the newest data

#### 9.10.3.3 getDataNormalized()

```
void Counter::getDataNormalized (
    std::function< double *(size_t, size_t)> array_out,
    bool rolling = true )
```

get countrate in Hz

the counts are normalized are copied to a newly allocated allocated memory, an the pointer to this location is returned. Invalid bins are replaced with NaNs.

##### Parameters

<i>array_out</i>	allocator callback for managed return values
<i>rolling</i>	if true, the returning array starts with the oldest data and goes up to the newest data

#### 9.10.3.4 getDataObject()

```
CounterData Counter::getDataObject (
    bool remove = false )
```

Fetch the most recent up to `n_values` bins, which have not been removed before.

This method allows atomic polling of bins, so each bin is guaranteed to be returned exactly once.

#### Parameters

<i>remove</i>	remove all fetched bins
---------------	-------------------------

#### Returns

a [CounterData](#) object, which contains all data of the fetches bins

### 9.10.3.5 `getDataTotalCounts()`

```
void Counter::getDataTotalCounts (
    std::function< uint64_t *(size_t)> array_out )
```

get the total amount of clicks per channel since the last clear including the currently integrating bin

### 9.10.3.6 `getIndex()`

```
void Counter::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

A vector of size `n_values` containing the time bins in ps.

### 9.10.3.7 `next_impl()`

```
bool Counter::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.10.3.8 on\_start()

```
void Counter::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.11 CounterData Class Reference

Helper object as return value for [Counter::getDataObject](#).

```
#include <Iterators.h>
```

### Public Member Functions

- [~CounterData](#) ( )
- void [getData](#) (std::function< int32\_t \*(size\_t, size\_t)> array\_out)  
*get the amount of clicks (or 0 if overflow occurs) per bin and per channel*
- void [getFrequency](#) (std::function< double \*(size\_t, size\_t)> array\_out, [timestamp\\_t](#) time\_scale=1000000000000)  
*get the counts normalized to the specified time scale*
- void [getDataNormalized](#) (std::function< double \*(size\_t, size\_t)> array\_out)  
*get the average rate of clicks per bin and per channel*
- void [getDataTotalCounts](#) (std::function< uint64\_t \*(size\_t)> array\_out)  
*get the total amount of clicks per channel since the last clear up to the most recent bin*
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)  
*get an index which corresponds to the timestamp of these bins*
- void [getTime](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)  
*get the timestamp of the bins since the last clear*
- void [getOverflowMask](#) (std::function< signed char \*(size\_t)> array\_out)  
*get if the bins were in overflow*
- void [getChannels](#) (std::function< [channel\\_t](#) \*(size\_t)> array\_out)  
*get the configured list of channels*

### Public Attributes

- const uint32\_t [size](#)  
*number of returned bins*
- const uint32\_t [dropped\\_bins](#)  
*number of bins which have been dropped because n\_bins has been exceeded, usually 0*
- const bool [overflow](#)  
*has anything been in overflow mode*

### 9.11.1 Detailed Description

Helper object as return value for [Counter::getDataObject](#).

This object stores the result of up to `n_values` bins.

### 9.11.2 Constructor & Destructor Documentation

#### 9.11.2.1 ~CounterData()

```
CounterData::~CounterData ( )
```

### 9.11.3 Member Function Documentation

#### 9.11.3.1 getChannels()

```
void CounterData::getChannels (
    std::function< channel_t *(size_t)> array_out )
```

get the configured list of channels

#### 9.11.3.2 getData()

```
void CounterData::getData (
    std::function< int32_t *(size_t, size_t)> array_out )
```

get the amount of clicks (or 0 if overflow occurs) per bin and per channel

Consider using `getFrequency` for explicit overflows. Alternatively, you may check overflow field and/or call `getOverflowMask` function.

#### 9.11.3.3 getDataNormalized()

```
void CounterData::getDataNormalized (
    std::function< double *(size_t, size_t)> array_out )
```

get the average rate of clicks per bin and per channel

#### 9.11.3.4 getDataTotalCounts()

```
void CounterData::getDataTotalCounts (
    std::function< uint64_t *(size_t)> array_out )
```

get the total amount of clicks per channel since the last clear up to the most recent bin

### 9.11.3.5 getFrequency()

```
void CounterData::getFrequency (
    std::function< double *(size_t, size_t)> array_out,
    timestamp_t time_scale = 1000000000000 )
```

get the counts normalized to the specified time scale

Bins in overflow mode are marked as NaN. The parameter `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`.

### 9.11.3.6 getIndex()

```
void CounterData::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

get an index which corresponds to the timestamp of these bins

### 9.11.3.7 getOverflowMask()

```
void CounterData::getOverflowMask (
    std::function< signed char *(size_t)> array_out )
```

get if the bins were in overflow

### 9.11.3.8 getTime()

```
void CounterData::getTime (
    std::function< timestamp_t *(size_t)> array_out )
```

get the timestamp of the bins since the last clear

## 9.11.4 Member Data Documentation

### 9.11.4.1 dropped\_bins

```
const uint32_t CounterData::dropped_bins
```

number of bins which have been dropped because `n_bins` has been exceeded, usually 0

### 9.11.4.2 overflow

```
const bool CounterData::overflow
```

has anything been in overflow mode

### 9.11.4.3 size

```
const uint32_t CounterData::size
```

number of returned bins

The documentation for this class was generated from the following file:

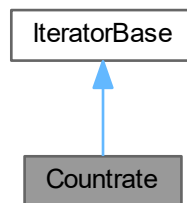
- [Iterators.h](#)

## 9.12 Countrate Class Reference

count rate on one or more channels

```
#include <Iterators.h>
```

Inheritance diagram for Countrate:



### Public Member Functions

- `Countrate` (`TimeTaggerBase` \*`tagger`, `std::vector`< `channel_t` > `channels`)  
*constructor of `Countrate`*
- `~Countrate` ()
- void `getData` (`std::function`< `double` \*(`size_t`)> `array_out`)  
*get the count rates*
- void `getCountsTotal` (`std::function`< `int64_t` \*(`size_t`)> `array_out`)  
*get the total amount of events*



## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`
- void `checkForAbort (T callback)`

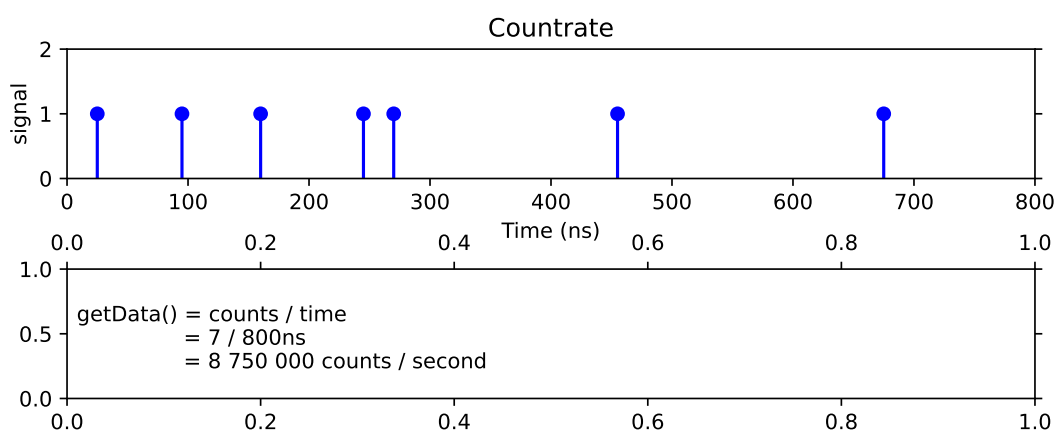
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`
- list of channels used by the iterator*
- bool `running`
- running state of the iterator*
- bool `autostart`
- Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`
- Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`
- Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`
- For internal use.*
- `std::atomic< bool > aborting`

## 9.12.1 Detailed Description

count rate on one or more channels



Measures the average count rate on one or more channels. Specifically, it counts incoming clicks and determines the time between the initial click and the latest click. The number of clicks divided by the time corresponds to the average countrate since the initial click.

## 9.12.2 Constructor & Destructor Documentation

### 9.12.2.1 Countrate()

```
Countrate::Countrate (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels )
```

constructor of [Countrate](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>channels</i>	the channels to count on

### 9.12.2.2 ~Countrate()

```
Countrate::~Countrate ( )
```

## 9.12.3 Member Function Documentation

### 9.12.3.1 clear\_impl()

```
void Countrate::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.12.3.2 getCountsTotal()

```
void Countrate::getCountsTotal (
    std::function< int64_t *(size_t)> array_out )
```

get the total amount of events

Returns the total amount of events per channel as an array.

### 9.12.3.3 getData()

```
void Countrate::getData (
    std::function< double *(size_t)> array_out )
```

get the count rates

Returns the average rate of events per second per channel as an array.

### 9.12.3.4 next\_impl()

```
bool Countrate::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.12.3.5 on\_start()

```
void Countrate::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.13 CustomLogger Class Reference

Helper class for setLogger.

```
#include <TimeTagger.h>
```

#### Public Member Functions

- [CustomLogger](#) ()
- virtual [~CustomLogger](#) ()
- void [enable](#) ()
- void [disable](#) ()
- virtual void [Log](#) (int level, const std::string &msg)=0

### 9.13.1 Detailed Description

Helper class for setLogger.

### 9.13.2 Constructor & Destructor Documentation

#### 9.13.2.1 CustomLogger()

```
CustomLogger::CustomLogger ( )
```

#### 9.13.2.2 ~CustomLogger()

```
virtual CustomLogger::~CustomLogger ( ) [virtual]
```

### 9.13.3 Member Function Documentation

#### 9.13.3.1 disable()

```
void CustomLogger::disable ( )
```

#### 9.13.3.2 enable()

```
void CustomLogger::enable ( )
```

#### 9.13.3.3 Log()

```
virtual void CustomLogger::Log (
    int level,
    const std::string & msg ) [pure virtual]
```

The documentation for this class was generated from the following file:

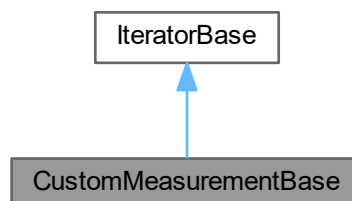
- [TimeTagger.h](#)

## 9.14 CustomMeasurementBase Class Reference

Helper class for custom measurements in Python and C#.

```
#include <Iterators.h>
```

Inheritance diagram for CustomMeasurementBase:



## Public Member Functions

- [~CustomMeasurementBase](#) () override
- void [register\\_channel](#) ([channel\\_t](#) channel)
- void [unregister\\_channel](#) ([channel\\_t](#) channel)
- void [finalize\\_init](#) ()
- bool [is\\_running](#) () const
- void [\\_lock](#) ()
- void [\\_unlock](#) ()

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Static Public Member Functions

- static void [stop\\_all\\_custom\\_measurements](#) ()

## Protected Member Functions

- [CustomMeasurementBase](#) ([TimeTaggerBase](#) \*[tagger](#))
- virtual bool [next\\_impl](#) (std::vector< [Tag](#) > &[incoming\\_tags](#), [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#))  
override  
*update iterator state*
- virtual void [next\\_impl\\_cs](#) (void \*[tags\\_ptr](#), [uint64\\_t](#) [num\\_tags](#), [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#))
- virtual void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) () override  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) () override  
*callback when the measurement class is stopped*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.14.1 Detailed Description

Helper class for custom measurements in Python and C#.

## 9.14.2 Constructor & Destructor Documentation

### 9.14.2.1 CustomMeasurementBase()

```
CustomMeasurementBase::CustomMeasurementBase (
    TimeTaggerBase * tagger ) [protected]
```

### 9.14.2.2 ~CustomMeasurementBase()

```
CustomMeasurementBase::~~CustomMeasurementBase ( ) [override]
```

## 9.14.3 Member Function Documentation

### 9.14.3.1 \_lock()

```
void CustomMeasurementBase::_lock ( )
```

### 9.14.3.2 \_unlock()

```
void CustomMeasurementBase::_unlock ( )
```

### 9.14.3.3 clear\_impl()

```
virtual void CustomMeasurementBase::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.14.3.4 finalize\_init()

```
void CustomMeasurementBase::finalize_init ( )
```

### 9.14.3.5 is\_running()

```
bool CustomMeasurementBase::is_running ( ) const
```

### 9.14.3.6 next\_impl()

```
virtual bool CustomMeasurementBase::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp\_t begin_time,
    timestamp\_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.



## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.14.3.7 next\_impl\_cs()**

```
virtual void CustomMeasurementBase::next_impl_cs (
    void * tags_ptr,
    uint64_t num_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [protected], [virtual]
```

**9.14.3.8 on\_start()**

```
virtual void CustomMeasurementBase::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.14.3.9 on\_stop()**

```
virtual void CustomMeasurementBase::on_stop ( ) [override], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.14.3.10 register\_channel()**

```
void CustomMeasurementBase::register_channel (
    channel_t channel )
```

**9.14.3.11 stop\_all\_custom\_measurements()**

```
static void CustomMeasurementBase::stop_all_custom_measurements ( ) [static]
```

### 9.14.3.12 unregister\_channel()

```
void CustomMeasurementBase::unregister_channel (
    channel_t channel )
```

The documentation for this class was generated from the following file:

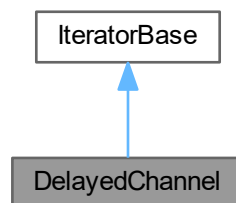
- [Iterators.h](#)

## 9.15 DelayedChannel Class Reference

a simple delayed queue

```
#include <Iterators.h>
```

Inheritance diagram for DelayedChannel:



### Public Member Functions

- [DelayedChannel](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, [timestamp\\_t](#) delay)  
*constructor of a [DelayedChannel](#)*
- [DelayedChannel](#) ([TimeTaggerBase](#) \*tagger, std::vector< [channel\\_t](#) > input\_channels, [timestamp\\_t](#) delay)  
*constructor of a [DelayedChannel](#) for delaying many channels at once*
- [~DelayedChannel](#) ()
- [channel\\_t](#) getChannel ()  
*the first new virtual channel*
- std::vector< [channel\\_t](#) > getChannels ()  
*the new virtual channels*
- void setDelay ([timestamp\\_t](#) delay)  
*set the delay time delay for the cloned tags in the virtual channels. A negative delay will delay all other events.*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`  
void `checkForAbort (T callback)`

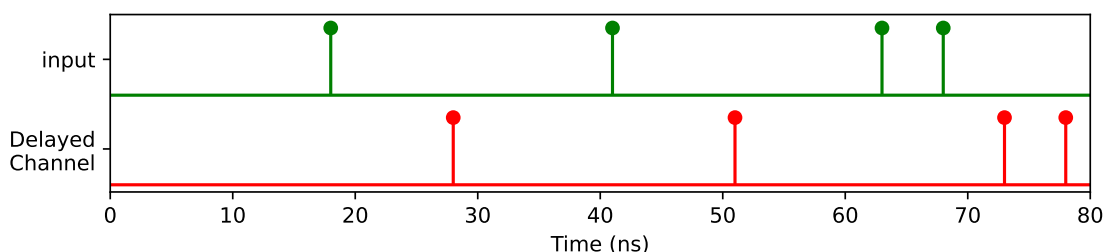
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.15.1 Detailed Description

a simple delayed queue



A simple first-in first-out queue of delayed event timestamps.

## 9.15.2 Constructor & Destructor Documentation

### 9.15.2.1 DelayedChannel() [1/2]

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

constructor of a [DelayedChannel](#)

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel which is delayed
<i>delay</i>	amount of time to delay

**9.15.2.2 DelayedChannel()** [2/2]

```
DelayedChannel::DelayedChannel (
    TimeTaggerBase * tagger,
    std::vector< channel_t > input_channels,
    timestamp_t delay )
```

constructor of a [DelayedChannel](#) for delaying many channels at once

This function is not exposed to Python/C#/Matlab/Labview

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channels</i>	channels which will be delayed
<i>delay</i>	amount of time to delay

**9.15.2.3 ~DelayedChannel()**

```
DelayedChannel::~~DelayedChannel ( )
```

**9.15.3 Member Function Documentation****9.15.3.1 getChannel()**

```
channel_t DelayedChannel::getChannel ( )
```

the first new virtual channel

This function returns the first of the new allocated virtual channels. It can be used now in any new iterator.

**9.15.3.2 getChannels()**

```
std::vector< channel_t > DelayedChannel::getChannels ( )
```

the new virtual channels

This function returns the new allocated virtual channels. It can be used now in any new iterator.

### 9.15.3.3 next\_impl()

```
bool DelayedChannel::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.15.3.4 on\_start()

```
void DelayedChannel::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.15.3.5 setDelay()

```
void DelayedChannel::setDelay (
    timestamp_t delay )
```

set the delay time delay for the cloned tags in the virtual channels. A negative delay will delay all other events.

Note: When the delay is the same or greater than the previous value all incoming tags will be visible at virtual channel. By applying a shorter delay time, the tags stored in the local buffer will be flushed and won't be visible in the virtual channel.

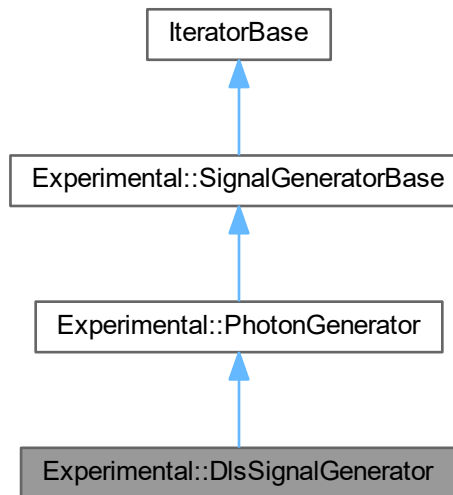
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.16 Experimental::DlsSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::DlsSignalGenerator:



### Public Member Functions

- [DlsSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double decay\_time, double countrate, [channel\\_t](#) output\_↔ channel=[CHANNEL\\_UNUSED](#), int32\_t seed=-1)  
*Construct a DLS event channel.*
- [DlsSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, std::vector< double > decay\_times, double countrate, [channel\\_t](#) output\_channel=[CHANNEL\\_UNUSED](#), int32\_t seed=-1)
- [~DlsSignalGenerator](#) ()
- unsigned int [get\\_N](#) ()

### Public Member Functions inherited from [Experimental::PhotonGenerator](#)

- [PhotonGenerator](#) ([TimeTaggerBase](#) \*tagger, double countrate, [channel\\_t](#) base\_channel, int32\_t seed=-1)  
*A generator for TimeTags arising from a laser driven process. [PhotonGenerator](#) should be used as the base class of a virtual class with a dedicated `get_intensity` function which models the relevant physical processes.*
- [~PhotonGenerator](#) ()
- void [finalize\\_init](#) ()
- void [set\\_T\\_PERIOD](#) ([timestamp\\_t](#) new\_T)
- [timestamp\\_t](#) [get\\_T\\_PERIOD](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*



## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- double [get\\_intensity](#) () override

## Protected Member Functions inherited from [Experimental::PhotonGenerator](#)

- void [initialize](#) (timestamp\_t initial\_time) override
- void [on\\_restart](#) (timestamp\_t restart\_time) override
- timestamp\_t [get\\_next](#) () override

## Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool is\_finished)

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [Experimental::PhotonGenerator](#)

- [timestamp\\_t](#) T\_PERIOD

## Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- std::unique\_ptr< [SignalGeneratorBaseImpl](#) > impl

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

## 9.16.1 Constructor & Destructor Documentation

### 9.16.1.1 DlsSignalGenerator() [1/2]

```
Experimental::DlsSignalGenerator::DlsSignalGenerator (
    TimeTaggerBase * tagger,
    double decay_time,
    double countrate,
    channel_t output_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

Construct a DLS event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a> .
<i>decay_time</i>	characteristic decay times (in seconds) for the g2 curve.
<i>countrate</i>	rate (in Hz) of Time Tags to be generated.
<i>output_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.16.1.2 DlsSignalGenerator() [2/2]

```
Experimental::DlsSignalGenerator::DlsSignalGenerator (
    TimeTaggerBase * tagger,
    std::vector< double > decay_times,
    double countrate,
    channel_t output_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

### 9.16.1.3 ~DlsSignalGenerator()

```
Experimental::DlsSignalGenerator::~~DlsSignalGenerator ( )
```

## 9.16.2 Member Function Documentation

### 9.16.2.1 get\_intensity()

```
double Experimental::DlsSignalGenerator::get_intensity ( ) [override], [protected], [virtual]
```

Implements [Experimental::PhotonGenerator](#).

### 9.16.2.2 get\_N()

```
unsigned int Experimental::DlsSignalGenerator::get_N ( )
```

The documentation for this class was generated from the following file:

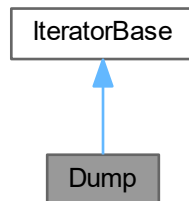
- [Iterators.h](#)

## 9.17 Dump Class Reference

dump all time tags to a file

```
#include <Iterators.h>
```

Inheritance diagram for Dump:



### Public Member Functions

- `Dump (TimeTaggerBase *tagger, std::string filename, int64_t max_tags, std::vector< channel_t > channels=std::vector< channel_t >())`  
*constructor of a `Dump` thread*
- `~Dump ()`

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase ()`  
*destructor, will unregister from the Time Tagger prior finalization.*
- void `start ()`  
*Starts or continues data acquisition.*
- void `startFor (timestamp_t capture_duration, bool clear=true)`  
*Starts or continues the data acquisition for the given duration.*
- bool `waitUntilFinished (int64_t timeout=-1)`  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- void `stop ()`  
*After calling this method, the measurement will stop processing incoming tags.*
- void `clear ()`  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort ()`  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning ()`  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration ()`  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration ()`  
*Fetches the overall configuration status of the measurement.*

**Protected Member Functions**

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*
- void `on_start` () override  
*callback when the measurement class is started*
- void `on_stop` () override  
*callback when the measurement class is stopped*

**Protected Member Functions inherited from `IteratorBase`**

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

**Additional Inherited Members****Protected Attributes inherited from `IteratorBase`**

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

### 9.17.1 Detailed Description

dump all time tags to a file

### 9.17.2 Constructor & Destructor Documentation

#### 9.17.2.1 Dump()

```
Dump::Dump (
    TimeTaggerBase * tagger,
    std::string filename,
    int64_t max_tags,
    std::vector< channel_t > channels = std::vector< channel_t >() )
```

constructor of a [Dump](#) thread

##### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>filename</i>	name of the file to dump to, must be encoded as UTF-8
<i>max_tags</i>	stop after this number of tags has been dumped. Negative values will dump forever
<i>channels</i>	channels which are dumped to the file (when empty or not passed all active channels are dumped)

#### 9.17.2.2 ~Dump()

```
Dump::~Dump ( )
```

### 9.17.3 Member Function Documentation

#### 9.17.3.1 clear\_impl()

```
void Dump::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.17.3.2 next\_impl()

```
bool Dump::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

**Parameters**

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.17.3.3 on\_start()**

```
void Dump::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.17.3.4 on\_stop()**

```
void Dump::on_stop ( ) [override], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

**9.18 Event Struct Reference**

Object for the return value of [Scope::getData](#).

```
#include <Iterators.h>
```

**Public Attributes**

- [timestamp\\_t](#) time
- [State](#) state

### 9.18.1 Detailed Description

Object for the return value of [Scope::getData](#).

### 9.18.2 Member Data Documentation

#### 9.18.2.1 state

[State](#) `Event::state`

#### 9.18.2.2 time

[timestamp\\_t](#) `Event::time`

The documentation for this struct was generated from the following file:

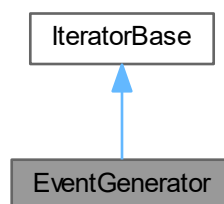
- [Iterators.h](#)

## 9.19 EventGenerator Class Reference

Generate predefined events in a virtual channel relative to a trigger event.

```
#include <Iterators.h>
```

Inheritance diagram for EventGenerator:



### Public Member Functions

- [EventGenerator](#) ([TimeTaggerBase](#) \*[tagger](#), [channel\\_t](#) trigger\_channel, std::vector< [timestamp\\_t](#) > pattern, uint64\_t trigger\_divider=1, uint64\_t divider\_offset=0, [channel\\_t](#) stop\_channel=[CHANNEL\\_UNUSED](#))  
*construct a event generator*
- [~EventGenerator](#) ()
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*



## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- template<typename T >
- void `checkForAbort (T callback)`

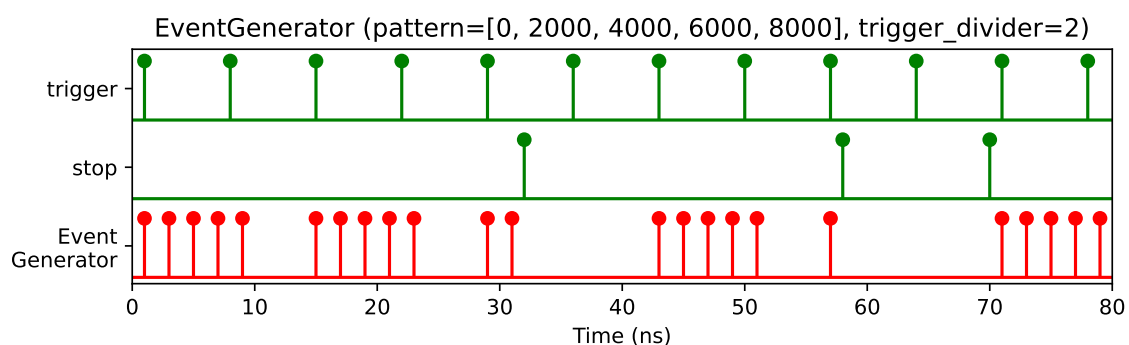
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`
- list of channels used by the iterator*
- bool `running`
- running state of the iterator*
- bool `autostart`
- Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`
- Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`
- Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`
- For internal use.*
- `std::atomic< bool > aborting`

## 9.19.1 Detailed Description

Generate predefined events in a virtual channel relative to a trigger event.



This iterator can be used to generate a predefined series of events, the pattern, relative to a trigger event on a defined channel. A `trigger_divider` can be used to fire the pattern not on every, but on every *n*'th trigger received. The `trigger_offset` can be used to select on which of the triggers the pattern will be generated when `trigger_divider` is greater than 1. To abort the pattern being generated, a `stop_channel` can be defined. In case it is the very same as the `trigger_channel`, the subsequent generated patterns will not overlap.

## 9.19.2 Constructor & Destructor Documentation

### 9.19.2.1 EventGenerator()

```
EventGenerator::EventGenerator (
    TimeTaggerBase * tagger,
    channel_t trigger_channel,
    std::vector< timestamp_t > pattern,
    uint64_t trigger_divider = 1,
    uint64_t divider_offset = 0,
    channel_t stop_channel = CHANNEL_UNUSED )
```

construct a event generator

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>trigger_channel</i>	trigger for generating the pattern
<i>pattern</i>	vector of time stamp generated relative to the trigger event
<i>trigger_divider</i>	establishes every how many trigger events a pattern is generated
<i>divider_offset</i>	the offset of the divided trigger when the pattern shall be emitted
<i>stop_channel</i>	channel on which a received event will stop all pending patterns from being generated

### 9.19.2.2 ~EventGenerator()

```
EventGenerator::~EventGenerator ( )
```

## 9.19.3 Member Function Documentation

### 9.19.3.1 clear\_impl()

```
void EventGenerator::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.19.3.2 getChannel()

```
channel_t EventGenerator::getChannel ( )
```

the new virtual channel

This function returns the new allocated virtual channel. It can be used now in any new iterator.

### 9.19.3.3 next\_impl()

```
bool EventGenerator::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.19.3.4 on\_start()

```
void EventGenerator::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

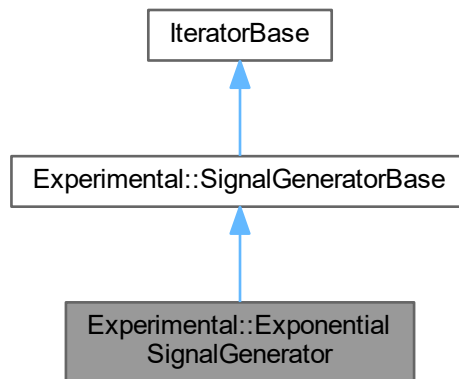
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.20 Experimental::ExponentialSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::ExponentialSignalGenerator:



### Public Member Functions

- [ExponentialSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double rate, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#), [int32\\_t](#) seed=-1)  
*Construct a exponential event channel.*
- [~ExponentialSignalGenerator](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
*the new virtual channel*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()

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

- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- void `initialize` (`timestamp_t` initial\_time) override
- `timestamp_t` `get_next` () override
- void `on_restart` (`timestamp_t` restart\_time) override

### Protected Member Functions inherited from `Experimental::SignalGeneratorBase`

- bool `next_impl` (`std::vector`< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `on_stop` () override  
*callback when the measurement class is stopped*
- bool `isProcessingFinished` ()
- void `set_processing_finished` (bool is\_finished)

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock`< `std::mutex` > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

## Additional Inherited Members

### Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- `std::unique_ptr< SignalGeneratorBaseImpl > impl`

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.20.1 Constructor & Destructor Documentation

### 9.20.1.1 ExponentialSignalGenerator()

```
Experimental::ExponentialSignalGenerator::ExponentialSignalGenerator (
    TimeTaggerBase * tagger,
    double rate,
    channel\_t base_channel = CHANNEL\_UNUSED,
    int32_t seed = -1 )
```

Construct a exponential event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>rate</i>	event rate in herz
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.20.1.2 ~ExponentialSignalGenerator()

```
Experimental::ExponentialSignalGenerator::~~ExponentialSignalGenerator ( )
```

## 9.20.2 Member Function Documentation

### 9.20.2.1 `get_next()`

```
timestamp_t Experimental::ExponentialSignalGenerator::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.20.2.2 `initialize()`

```
void Experimental::ExponentialSignalGenerator::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.20.2.3 `on_restart()`

```
void Experimental::ExponentialSignalGenerator::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.21 FastBinning Class Reference

Helper class for fast division with a constant divisor.

```
#include <Iterators.h>
```

### Public Types

- enum class [Mode](#) {  
[ConstZero](#) , [Dividend](#) , [PowerOfTwo](#) , [FixedPoint\\_32](#) ,  
[FixedPoint\\_64](#) , [Divide\\_32](#) , [Divide\\_64](#) }

### Public Member Functions

- [FastBinning](#) ()
- [FastBinning](#) (uint64\_t divisor, uint64\_t max\_duration\_)
- template<[Mode](#) mode>  
 uint64\_t [divide](#) (uint64\_t duration) const
- [Mode](#) [getMode](#) () const

### 9.21.1 Detailed Description

Helper class for fast division with a constant divisor.

It chooses the method on initialization time and precompile the evaluation functions for all methods.

## 9.21.2 Member Enumeration Documentation

### 9.21.2.1 `Mode`

```
enum class FastBinning::Mode [strong]
```



## Enumerator

ConstZero	
Dividend	
PowerOfTwo	
FixedPoint_32	
FixedPoint_64	
Divide_32	
Divide_64	

### 9.21.3 Constructor & Destructor Documentation

#### 9.21.3.1 FastBinning() [1/2]

```
FastBinning::FastBinning ( ) [inline]
```

#### 9.21.3.2 FastBinning() [2/2]

```
FastBinning::FastBinning (
    uint64_t divisor,
    uint64_t max_duration_ )
```

### 9.21.4 Member Function Documentation

#### 9.21.4.1 divide()

```
template<Mode mode>
uint64_t FastBinning::divide (
    uint64_t duration ) const [inline]
```

#### 9.21.4.2 getMode()

```
Mode FastBinning::getMode ( ) const [inline]
```

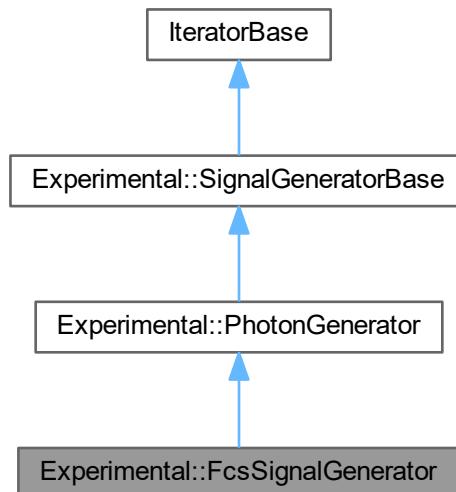
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.22 Experimental::FcsSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::FcsSignalGenerator:



### Public Member Functions

- [FcsSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double correlation\_time, double N\_focus, double countrate, [channel\\_t](#) output\_channel=[CHANNEL\\_UNUSED](#), [int32\\_t](#) seed=-1)  
Construct an FCS event channel.
- [~FcsSignalGenerator](#) ()
- unsigned int [get\\_N](#) ()
- void [set\\_boundary\\_limit](#) (double new\_boundary)

### Public Member Functions inherited from [Experimental::PhotonGenerator](#)

- [PhotonGenerator](#) ([TimeTaggerBase](#) \*tagger, double countrate, [channel\\_t](#) base\_channel, [int32\\_t](#) seed=-1)  
A generator for TimeTags arising from a laser driven process. [PhotonGenerator](#) should be used as the base class of a virtual class with a dedicated *get\_intensity* function which models the relevant physical processes.
- [~PhotonGenerator](#) ()
- void [finalize\\_init](#) ()
- void [set\\_T\\_PERIOD](#) ([timestamp\\_t](#) new\_T)
- [timestamp\\_t](#) [get\\_T\\_PERIOD](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) [getChannel](#) ()  
the new virtual channel

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- double [get\\_intensity](#) () override

## Protected Member Functions inherited from [Experimental::PhotonGenerator](#)

- void [initialize](#) (timestamp\_t initial\_time) override
- void [on\\_restart](#) (timestamp\_t restart\_time) override
- timestamp\_t [get\\_next](#) () override

## Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool is\_finished)

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [Experimental::PhotonGenerator](#)

- [timestamp\\_t](#) T\_PERIOD

## Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- std::unique\_ptr< [SignalGeneratorBaseImpl](#) > impl

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

## 9.22.1 Constructor & Destructor Documentation

### 9.22.1.1 FcsSignalGenerator()

```
Experimental::FcsSignalGenerator::FcsSignalGenerator (
    TimeTaggerBase * tagger,
    double correlation_time,
    double N_focus,
    double countrate,
    channel_t output_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

Construct an FCS event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a> .
<i>correlation_time</i>	characteristic correlation time in the exponential g2 curve.
<i>countrate</i>	rate (in Hz) of Time Tags to be generated.
<i>N_focus</i>	the average number of particles in the laser focus.
<i>output_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.22.1.2 ~FcsSignalGenerator()

```
Experimental::FcsSignalGenerator::~~FcsSignalGenerator ( )
```

## 9.22.2 Member Function Documentation

### 9.22.2.1 get\_intensity()

```
double Experimental::FcsSignalGenerator::get_intensity ( ) [override], [protected], [virtual]
```

Implements [Experimental::PhotonGenerator](#).

### 9.22.2.2 get\_N()

```
unsigned int Experimental::FcsSignalGenerator::get_N ( )
```

### 9.22.2.3 set\_boundary\_limit()

```
void Experimental::FcsSignalGenerator::set_boundary_limit (
    double new_boundary )
```

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.23 FileReader Class Reference

Reads tags from the disk files, which has been created by [FileWriter](#).

```
#include <Iterators.h>
```

### Public Member Functions

- [FileReader](#) (std::vector< std::string > filenames)  
*Creates a file reader with the given filename.*
- [FileReader](#) (const std::string &filename)  
*Creates a file reader with the given filename.*
- [~FileReader](#) ()
- bool [hasData](#) ()  
*Checks if there are still events in the [FileReader](#).*
- [TimeTagStreamBuffer](#) [getData](#) (uint64\_t n\_events)  
*Fetches and delete the next tags from the internal buffer.*
- bool [getDataRaw](#) (std::vector< [Tag](#) > &tag\_buffer)  
*Low level file reading.*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the Time Tagger object, which was serialized in the current file.*
- std::vector< [channel\\_t](#) > [getChannelList](#) ()  
*Fetches channels from the input file.*
- std::string [getLastMarker](#) ()  
*return the last processed marker from the file.*

### 9.23.1 Detailed Description

Reads tags from the disk files, which has been created by [FileWriter](#).

Its usage is compatible with the [TimeTagStream](#).

### 9.23.2 Constructor & Destructor Documentation

#### 9.23.2.1 FileReader() [1/2]

```
FileReader::FileReader (
    std::vector< std::string > filenames )
```

Creates a file reader with the given filename.

The file reader automatically continues to read split [FileWriter](#) Streams In case multiple filenames are given, the files will be read in successively.

#### Parameters

<i>filenames</i>	list of files to read, must be encoded as UTF-8
------------------	---

### 9.23.2.2 FileReader() [2/2]

```
FileReader::FileReader (
    const std::string & filename )
```

Creates a file reader with the given filename.

The file reader automatically continues to read split [FileWriter](#) Streams

#### Parameters

<i>filename</i>	file to read, must be encoded as UTF-8
-----------------	--

### 9.23.2.3 ~FileReader()

```
FileReader::~FileReader ( )
```

## 9.23.3 Member Function Documentation

### 9.23.3.1 getChannelList()

```
std::vector< channel_t > FileReader::getChannelList ( )
```

Fetches channels from the input file.

#### Returns

a vector of channels from the input file.

### 9.23.3.2 getConfiguration()

```
std::string FileReader::getConfiguration ( )
```

Fetches the overall configuration status of the Time Tagger object, which was serialized in the current file.

#### Returns

a JSON serialized string with all configuration and status flags.

### 9.23.3.3 getData()

```
TimeTagStreamBuffer FileReader::getData (
    uint64_t n_events )
```

Fetches and delete the next tags from the internal buffer.

Every tag is returned exactly once. If less than `n_events` are returned, the reader is at the end-of-files.

**Parameters**

<i>n_events</i>	maximum amount of elements to fetch
-----------------	-------------------------------------

**Returns**

a [TimeTagStreamBuffer](#) with up to *n\_events* events

**9.23.3.4 getDataRaw()**

```
bool FileReader::getDataRaw (
    std::vector< Tag > & tag_buffer )
```

Low level file reading.

This function will return the next non-empty buffer in a raw format.

**Parameters**

<i>tag_buffer</i>	a buffer, which will be filled with the new events
-------------------	--

**Returns**

true if fetching the data was successfully

**9.23.3.5 getLastMarker()**

```
std::string FileReader::getLastMarker ( )
```

return the last processed marker from the file.

**Returns**

the last marker from the file

**9.23.3.6 hasData()**

```
bool FileReader::hasData ( )
```

Checks if there are still events in the [FileReader](#).

**Returns**

false if no more events can be read from this [FileReader](#)

The documentation for this class was generated from the following file:

- [Iterators.h](#)

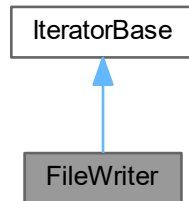


## 9.24 FileWriter Class Reference

compresses and stores all time tags to a file

```
#include <Iterators.h>
```

Inheritance diagram for FileWriter:



### Public Member Functions

- `FileWriter (TimeTaggerBase *tagger, const std::string &filename, std::vector< channel_t > channels)`  
*constructor of a [FileWriter](#)*
- `~FileWriter ()`
- `void split (const std::string &new_filename="")`  
*Close the current file and create a new one.*
- `void setMaxFileSize (uint64_t max_file_size)`  
*Set the maximum file size on disk when the automatic split shall happen.*
- `uint64_t getMaxFileSize ()`  
*fetches the maximum file size. Please see setMaxFileSize for more details.*
- `uint64_t getTotalEvents ()`  
*queries the total amount of events stored in all files*
- `uint64_t getTotalSize ()`  
*queries the total amount of bytes stored in all files*
- `void setMarker (const std::string &marker)`  
*writes a marker in the file. While parsing the file, the last marker can be extracted again.*

### Public Member Functions inherited from [IteratorBase](#)

- `virtual ~IteratorBase ()`  
*destructor, will unregister from the Time Tagger prior finalization.*
- `void start ()`  
*Starts or continues data acquisition.*
- `void startFor (timestamp_t capture_duration, bool clear=true)`  
*Starts or continues the data acquisition for the given duration.*
- `bool waitUntilFinished (int64_t timeout=-1)`  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- `void stop ()`

- *After calling this method, the measurement will stop processing incoming tags.*
- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (`std::vector< Tag >` &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear *Iterator* state.*
- void `on_start` () override  
*callback when the measurement class is started*
- void `on_stop` () override  
*callback when the measurement class is stopped*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- `template<typename T >`  
void `checkForAbort` (T callback)

## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the `finishInitialization` callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.24.1 Detailed Description

compresses and stores all time tags to a file

### 9.24.2 Constructor & Destructor Documentation

#### 9.24.2.1 `FileWriter()`

```
FileWriter::FileWriter (
    TimeTaggerBase * tagger,
    const std::string & filename,
    std::vector< channel\_t > channels )
```

constructor of a [FileWriter](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>filename</i>	name of the file to store to, must be encoded as UTF-8
<i>channels</i>	channels which are stored to the file

#### 9.24.2.2 `~FileWriter()`

```
FileWriter::~FileWriter ( )
```

### 9.24.3 Member Function Documentation

#### 9.24.3.1 `clear_impl()`

```
void FileWriter::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.24.3.2 getMaxFileSize()

```
uint64_t FileWriter::getMaxFileSize ( )
```

fetches the maximum file size. Please see [setMaxFileSize](#) for more details.

##### Returns

the maximum file size in bytes

#### 9.24.3.3 getTotalEvents()

```
uint64_t FileWriter::getTotalEvents ( )
```

queries the total amount of events stored in all files

##### Returns

the total amount of events stored

#### 9.24.3.4 getTotalSize()

```
uint64_t FileWriter::getTotalSize ( )
```

queries the total amount of bytes stored in all files

##### Returns

the total amount of bytes stored

#### 9.24.3.5 next\_impl()

```
bool FileWriter::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp\_t begin_time,
    timestamp\_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.24.3.6 on\_start()**

```
void FileWriter::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.24.3.7 on\_stop()**

```
void FileWriter::on_stop ( ) [override], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.24.3.8 setMarker()**

```
void FileWriter::setMarker (
    const std::string & marker )
```

writes a marker in the file. While parsing the file, the last marker can be extracted again.

## Parameters

<i>marker</i>	the marker to write into the file
---------------	-----------------------------------

**9.24.3.9 setMaxFileSize()**

```
void FileWriter::setMaxFileSize (
    uint64_t max_file_size )
```

Set the maximum file size on disk when the automatic split shall happen.

#### Note

This is a rough limit, the actual file might be larger by one block.

#### Parameters

<i>max_file_size</i>	new maximum file size in bytes
----------------------	--------------------------------

### 9.24.3.10 split()

```
void FileWriter::split (
    const std::string & new_filename = "" )
```

Close the current file and create a new one.

#### Parameters

<i>new_filename</i>	filename of the new file. If empty, the old one will be used.
---------------------	---

The documentation for this class was generated from the following file:

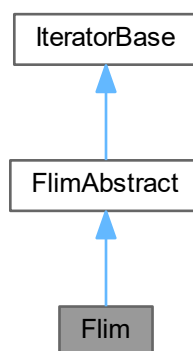
- [Iterators.h](#)

## 9.25 Flim Class Reference

Fluorescence lifetime imaging.

```
#include <Iterators.h>
```

Inheritance diagram for Flim:



## Public Member Functions

- [Flim](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, [channel\\_t](#) click\_channel, [channel\\_t](#) pixel\_begin\_channel, [uint32\\_t](#) n\_pixels, [uint32\\_t](#) n\_bins, [timestamp\\_t](#) binwidth, [channel\\_t](#) pixel\_end\_channel=CHANNEL\_UNUSED, [channel\\_t](#) frame\_begin\_channel=CHANNEL\_UNUSED, [uint32\\_t](#) finish\_after\_outputframe=0, [uint32\\_t](#) n\_frame\_average=1, [bool](#) pre\_initialize=true)  
construct a *Flim* measurement with a variety of high-level functionality
- [~Flim](#) ()  
initializes and starts measuring this *Flim* measurement
- [void initialize](#) ()  
initializes and starts measuring this *Flim* measurement
- [void getReadyFrame](#) (std::function< [uint32\\_t](#) \*([size\\_t](#), [size\\_t](#))> array\_out, [int32\\_t](#) index=-1)  
obtain for each pixel the histogram for the given frame index
- [void getReadyFrameIntensity](#) (std::function< [float](#) \*([size\\_t](#))> array\_out, [int32\\_t](#) index=-1)  
obtain an array of the pixel intensity of the given frame index
- [void getCurrentFrame](#) (std::function< [uint32\\_t](#) \*([size\\_t](#), [size\\_t](#))> array\_out)  
obtain for each pixel the histogram for the frame currently active
- [void getCurrentFrameIntensity](#) (std::function< [float](#) \*([size\\_t](#))> array\_out)  
obtain the array of the pixel intensities of the frame currently active
- [void getSummedFrames](#) (std::function< [uint32\\_t](#) \*([size\\_t](#), [size\\_t](#))> array\_out, [bool](#) only\_ready\_frames=true, [bool](#) clear\_summed=false)  
obtain for each pixel the histogram from all frames acquired so far
- [void getSummedFramesIntensity](#) (std::function< [float](#) \*([size\\_t](#))> array\_out, [bool](#) only\_ready\_frames=true, [bool](#) clear\_summed=false)  
obtain the array of the pixel intensities from all frames acquired so far
- [FlimFrameInfo getReadyFrameEx](#) ([int32\\_t](#) index=-1)  
obtain a frame information object, for the given frame index
- [FlimFrameInfo getCurrentFrameEx](#) ()  
obtain a frame information object, for the currently active frame
- [FlimFrameInfo getSummedFramesEx](#) ([bool](#) only\_ready\_frames=true, [bool](#) clear\_summed=false)  
obtain a frame information object, that represents the sum of all frames acquired so far.
- [uint32\\_t getFramesAcquired](#) () const  
total number of frames completed so far
- [void getIndex](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#))> array\_out)  
a vector of size n\_bins containing the time bins in ps

Public Member Functions inherited from [FlimAbstract](#)

- [FlimAbstract](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, [channel\\_t](#) click\_channel, [channel\\_t](#) pixel\_begin\_channel, [uint32\\_t](#) n\_pixels, [uint32\\_t](#) n\_bins, [timestamp\\_t](#) binwidth, [channel\\_t](#) pixel\_end\_channel=CHANNEL\_UNUSED, [channel\\_t](#) frame\_begin\_channel=CHANNEL\_UNUSED, [uint32\\_t](#) finish\_after\_outputframe=0, [uint32\\_t](#) n\_frame\_average=1, [bool](#) pre\_initialize=true)  
construct a *FlimAbstract* object, *Flim* and *FlimBase* classes inherit from it
- [~FlimAbstract](#) ()
- [bool isAcquiring](#) () const  
tells if the data acquisition has finished reaching finish\_after\_outputframe

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- void [on\\_frame\\_end](#) () override final
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- uint32\_t [get\\_ready\\_index](#) (int32\_t index)
- virtual void [frameReady](#) (uint32\_t frame\_number, std::vector< uint32\_t > &data, std::vector< timestamp\_t > &pixel\_begin\_times, std::vector< timestamp\_t > &pixel\_end\_times, timestamp\_t frame\_begin\_time, timestamp\_t frame\_end\_time)

## Protected Member Functions inherited from [FlimAbstract](#)

- template<[FastBinning::Mode](#) bin\_mode>  
void [process\\_tags](#) (const std::vector< [Tag](#) > &incoming\_tags)
- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*



## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Protected Attributes

- std::vector< std::vector< uint32\_t > > [back\\_frames](#)
- std::vector< std::vector< [timestamp\\_t](#) > > [frame\\_begins](#)
- std::vector< std::vector< [timestamp\\_t](#) > > [frame\\_ends](#)
- std::vector< uint32\_t > [pixels\\_completed](#)
- std::vector< uint32\_t > [summed\\_frames](#)
- std::vector< [timestamp\\_t](#) > [accum\\_diffs](#)
- uint32\_t [captured\\_frames](#)
- uint32\_t [total\\_frames](#)
- int32\_t [last\\_frame](#)
- std::mutex [swap\\_chain\\_lock](#)

## Protected Attributes inherited from [FlimAbstract](#)

- const [channel\\_t](#) [start\\_channel](#)
- const [channel\\_t](#) [click\\_channel](#)
- const [channel\\_t](#) [pixel\\_begin\\_channel](#)
- const uint32\_t [n\\_pixels](#)
- const uint32\_t [n\\_bins](#)
- const [timestamp\\_t](#) [binwidth](#)
- const [channel\\_t](#) [pixel\\_end\\_channel](#)
- const [channel\\_t](#) [frame\\_begin\\_channel](#)

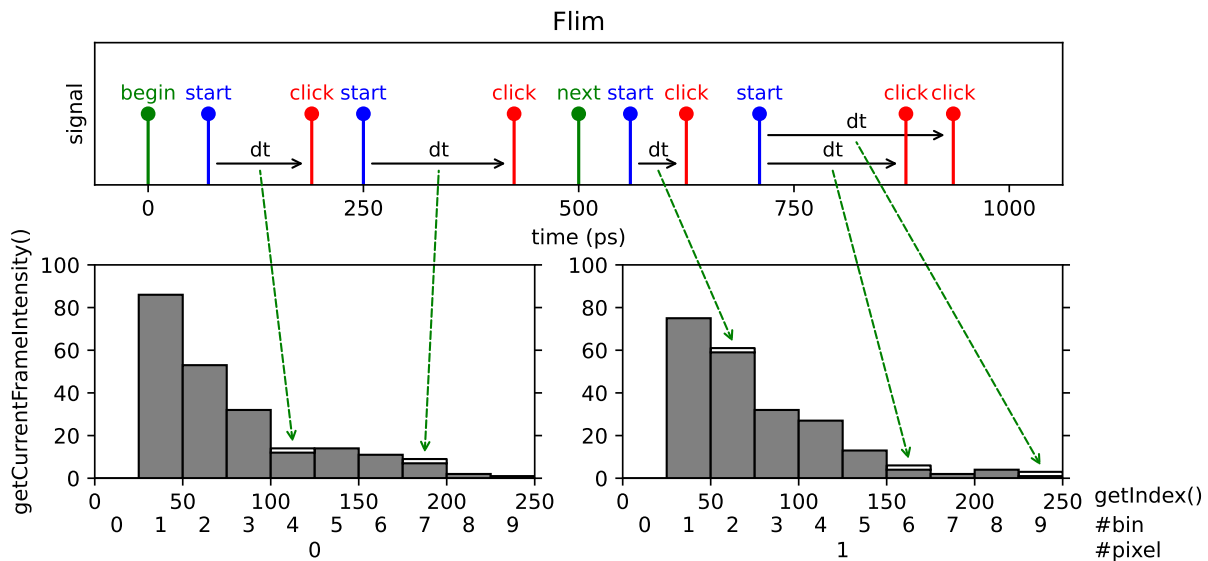
- const uint32\_t `finish_after_outputframe`
- const uint32\_t `n_frame_average`
- const `timestamp_t` `time_window`
- `timestamp_t` `current_frame_begin`
- `timestamp_t` `current_frame_end`
- bool `acquiring` {}
- bool `frame_acquisition` {}
- bool `pixel_acquisition` {}
- uint32\_t `pixels_processed` {}
- uint32\_t `frames_completed` {}
- uint32\_t `ticks` {}
- size\_t `data_base` {}
- std::vector< uint32\_t > `frame`
- std::vector< `timestamp_t` > `pixel_begins`
- std::vector< `timestamp_t` > `pixel_ends`
- std::deque< `timestamp_t` > `previous_starts`
- `FastBinning` `binner`
- std::recursive\_mutex `acquisition_lock`
- bool `initialized`

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

#### 9.25.1 Detailed Description

Fluorescence lifetime imaging.



Successively acquires  $n$  histograms (one for each pixel in the image), where each histogram is determined by the number of bins and the binwidth. Clicks that fall outside the histogram range are ignored.

Fluorescence-lifetime imaging microscopy or [Flim](#) is an imaging technique for producing an image based on the differences in the exponential decay rate of the fluorescence from a fluorescent 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 pulse of light, the time-resolved fluorescence will decay exponentially.

## 9.25.2 Constructor & Destructor Documentation

### 9.25.2.1 Flim()

```
Flim::Flim (
    TimeTaggerBase * tagger,
    channel_t start_channel,
    channel_t click_channel,
    channel_t pixel_begin_channel,
    uint32_t n_pixels,
    uint32_t n_bins,
    timestamp_t binwidth,
    channel_t pixel_end_channel = CHANNEL_UNUSED,
    channel_t frame_begin_channel = CHANNEL_UNUSED,
    uint32_t finish_after_outputframe = 0,
    uint32_t n_frame_average = 1,
    bool pre_initialize = true )
```

construct a [Flim](#) measurement with a variety of high-level functionality

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>start_channel</i>	channel on which start clicks are received for the time differences histogramming
<i>click_channel</i>	channel on which clicks are received for the time differences histogramming
<i>pixel_begin_channel</i>	start of a pixel (histogram)

## Parameters

<i>n_pixels</i>	number of pixels (histograms) of one frame
<i>n_bins</i>	number of histogram bins for each pixel
<i>binwidth</i>	bin size in picoseconds
<i>pixel_end_channel</i>	end marker of a pixel - incoming clicks on the click_channel will be ignored afterwards
<i>frame_begin_channel</i>	(optional) start the frame, or reset the pixel index
<i>finish_after_outputframe</i>	(optional) sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0 (default value), one frame is stored and the measurement runs continuously.
<i>n_frame_average</i>	(optional) average multiple input frames into one output frame, default: 1
<i>pre_initialize</i>	(optional) initializes the measurement on constructing.

## 9.25.2.2 ~Flim()

```
Flim::~~Flim ( )
```

## 9.25.3 Member Function Documentation

## 9.25.3.1 clear\_impl()

```
void Flim::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

## 9.25.3.2 frameReady()

```
virtual void Flim::frameReady (
    uint32_t frame_number,
    std::vector< uint32_t > & data,
    std::vector< timestamp_t > & pixel_begin_times,
    std::vector< timestamp_t > & pixel_end_times,
    timestamp_t frame_begin_time,
    timestamp_t frame_end_time ) [protected], [virtual]
```

## 9.25.3.3 get\_ready\_index()

```
uint32_t Flim::get_ready_index (
    int32_t index ) [protected]
```

#### 9.25.3.4 getCurrentFrame()

```
void Flim::getCurrentFrame (
    std::function< uint32_t *(size_t, size_t)> array_out )
```

obtain for each pixel the histogram for the frame currently active

This function returns the histograms for all pixels of the currently active frame

#### 9.25.3.5 getCurrentFrameEx()

```
FlimFrameInfo Flim::getCurrentFrameEx ( )
```

obtain a frame information object, for the currently active frame

This function returns the frame information object for the currently active frame

#### 9.25.3.6 getCurrentFrameIntensity()

```
void Flim::getCurrentFrameIntensity (
    std::function< float *(size_t)> array_out )
```

obtain the array of the pixel intensities of the frame currently active

This function 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.

#### 9.25.3.7 getFramesAcquired()

```
uint32_t Flim::getFramesAcquired ( ) const [inline]
```

total number of frames completed so far

This function returns the amount of frames that have been completed so far, since the creation / last clear of the object.

#### 9.25.3.8 getIndex()

```
void Flim::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

a vector of size n\_bins containing the time bins in ps

This function returns a vector of size n\_bins containing the time bins in ps.

#### 9.25.3.9 getReadyFrame()

```
void Flim::getReadyFrame (
    std::function< uint32_t *(size_t, size_t)> array_out,
    int32_t index = -1 )
```

obtain for each pixel the histogram for the given frame index

This function returns the histograms for all pixels according to the frame index given. If the index is -1, it will return the last frame, which has been completed. When finish\_after\_outputframe is 0, the index value must be -1. If index >= finish\_after\_outputframe, it will throw an error.

## Parameters

<i>array_out</i>	callback for the array output allocation
<i>index</i>	index of the frame to be obtained. if -1, the last frame which has been completed is returned

**9.25.3.10 getReadyFrameEx()**

```
FlimFrameInfo Flim::getReadyFrameEx (
    int32_t index = -1 )
```

obtain a frame information object, for the given frame index

This function returns a frame information object according to the index given. If the index is -1, it will return the last completed frame. When finish\_after\_outputframe is 0, index must be -1. If index >= finish\_after\_outputframe, it will throw an error.

## Parameters

<i>index</i>	index of the frame to be obtained. if -1, last completed frame will be returned
--------------	---

**9.25.3.11 getReadyFrameIntensity()**

```
void Flim::getReadyFrameIntensity (
    std::function< float *(size_t)> array_out,
    int32_t index = -1 )
```

obtain an array of the pixel intensity of the given frame index

This function returns the intensities according to the frame index given. If the index is -1, it will return the intensity of the last frame, which has been completed. When finish\_after\_outputframe is 0, the index value must be -1. If index >= finish\_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.

## Parameters

<i>array_out</i>	callback for the array output allocation
<i>index</i>	index of the frame to be obtained. if -1, the last frame which has been completed is returned

**9.25.3.12 getSummedFrames()**

```
void Flim::getSummedFrames (
    std::function< uint32_t *(size_t, size_t)> array_out,
    bool only_ready_frames = true,
    bool clear_summed = false )
```

obtain for each pixel the histogram from all frames acquired so far

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

## Parameters

<i>array_out</i>	callback for the array output allocation
<i>only_ready_frames</i>	if true, only the finished frames are added. On false, the currently active frame is aggregated.
<i>clear_summed</i>	if true, the summed frames memory will be cleared.

**9.25.3.13 getSummedFramesEx()**

```
FlimFrameInfo Flim::getSummedFramesEx (
    bool only_ready_frames = true,
    bool clear_summed = false )
```

obtain a frame information object, that represents the sum of all frames acquired so for.

This function returns the frame information object that represents the sum of all acquired frames.

## Parameters

<i>only_ready_frames</i>	if true only the finished frames are added. On false, the currently active is aggregated.
<i>clear_summed</i>	if true, the summed frames memory will be reset and all frames stored prior will be unaccounted in the future.

**9.25.3.14 getSummedFramesIntensity()**

```
void Flim::getSummedFramesIntensity (
    std::function< float *(size_t)> array_out,
    bool only_ready_frames = true,
    bool clear_summed = false )
```

obtain the array of the pixel intensities from all frames acquired so far

The pixel intensity is the number of counts within the pixel divided by the integration time.

This function returns the intensities of all pixels summed over all acquired frames.

## Parameters

<i>array_out</i>	callback for the array output allocation
<i>only_ready_frames</i>	if true only the finished frames are added. On false, the currently active frame is aggregated.
<i>clear_summed</i>	if true, the summed frames memory will be cleared.

**9.25.3.15 initialize()**

```
void Flim::initialize ( )
```

initializes and starts measuring this [Flim](#) measurement

This function initializes the [Flim](#) measurement and starts executing it. It does nothing if preinitialized in the constructor is set to true.



### 9.25.3.16 on\_frame\_end()

```
void Flim::on_frame_end ( ) [final], [override], [protected], [virtual]
```

Implements [FlimAbstract](#).

## 9.25.4 Member Data Documentation

### 9.25.4.1 accum\_diffs

```
std::vector<timestamp_t> Flim::accum_diffs [protected]
```

### 9.25.4.2 back\_frames

```
std::vector<std::vector<uint32_t> > Flim::back_frames [protected]
```

### 9.25.4.3 captured\_frames

```
uint32_t Flim::captured_frames [protected]
```

### 9.25.4.4 frame\_begins

```
std::vector<std::vector<timestamp_t> > Flim::frame_begins [protected]
```

### 9.25.4.5 frame\_ends

```
std::vector<std::vector<timestamp_t> > Flim::frame_ends [protected]
```

### 9.25.4.6 last\_frame

```
int32_t Flim::last_frame [protected]
```

### 9.25.4.7 pixels\_completed

```
std::vector<uint32_t> Flim::pixels_completed [protected]
```

### 9.25.4.8 summed\_frames

```
std::vector<uint32_t> Flim::summed_frames [protected]
```

#### 9.25.4.9 swap\_chain\_lock

```
std::mutex Flim::swap_chain_lock [protected]
```

#### 9.25.4.10 total\_frames

```
uint32_t Flim::total_frames [protected]
```

The documentation for this class was generated from the following file:

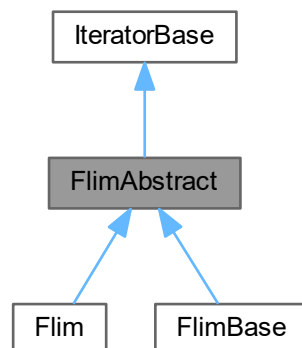
- [Iterators.h](#)

## 9.26 FlimAbstract Class Reference

Interface for FLIM measurements, [Flim](#) and [FlimBase](#) classes inherit from it.

```
#include <Iterators.h>
```

Inheritance diagram for FlimAbstract:



### Public Member Functions

- [FlimAbstract](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, [channel\\_t](#) click\_channel, [channel\\_t](#) pixel\_begin\_channel, [uint32\\_t](#) n\_pixels, [uint32\\_t](#) n\_bins, [timestamp\\_t](#) binwidth, [channel\\_t](#) pixel\_end\_channel=CHANNEL\_UNUSED, [channel\\_t](#) frame\_begin\_channel=CHANNEL\_UNUSED, [uint32\\_t](#) finish\_after\_outputframe=0, [uint32\\_t](#) n\_frame\_average=1, [bool](#) pre\_initialize=true)

*construct a [FlimAbstract](#) object, [Flim](#) and [FlimBase](#) classes inherit from it*

- [~FlimAbstract](#) ()
- [bool](#) [isAcquiring](#) () const

*tells if the data acquisition has finished reaching finish\_after\_outputframe*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- template<FastBinning::Mode bin\_mode>  
void [process\\_tags](#) (const std::vector< [Tag](#) > &incoming\_tags)  
*update iterator state*
- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*clear [Iterator](#) state.*
- void [clear\\_impl](#) () override  
*callback when the measurement class is started*
- virtual void [on\\_frame\\_end](#) ()=0

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*

- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- `template<typename T >`  
void `checkForAbort` (T callback)

### Protected Attributes

- const `channel_t` `start_channel`
- const `channel_t` `click_channel`
- const `channel_t` `pixel_begin_channel`
- const `uint32_t` `n_pixels`
- const `uint32_t` `n_bins`
- const `timestamp_t` `binwidth`
- const `channel_t` `pixel_end_channel`
- const `channel_t` `frame_begin_channel`
- const `uint32_t` `finish_after_outputframe`
- const `uint32_t` `n_frame_average`
- const `timestamp_t` `time_window`
- `timestamp_t` `current_frame_begin`
- `timestamp_t` `current_frame_end`
- bool `acquiring` {}
- bool `frame_acquisition` {}
- bool `pixel_acquisition` {}
- `uint32_t` `pixels_processed` {}
- `uint32_t` `frames_completed` {}
- `uint32_t` `ticks` {}
- `size_t` `data_base` {}
- `std::vector< uint32_t >` `frame`
- `std::vector< timestamp_t >` `pixel_begins`
- `std::vector< timestamp_t >` `pixel_ends`
- `std::deque< timestamp_t >` `previous_starts`
- `FastBinning` `binner`
- `std::recursive_mutex` `acquisition_lock`
- bool `initialized`

## Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.26.1 Detailed Description

Interface for FLIM measurements, [Flim](#) and [FlimBase](#) classes inherit from it.

### 9.26.2 Constructor & Destructor Documentation

#### 9.26.2.1 FlimAbstract()

```
FlimAbstract::FlimAbstract (
    TimeTaggerBase * tagger,
    channel\_t start_channel,
    channel\_t click_channel,
    channel\_t pixel_begin_channel,
    uint32_t n_pixels,
    uint32_t n_bins,
    timestamp_t binwidth,
    channel\_t pixel_end_channel = CHANNEL\_UNUSED,
    channel\_t frame_begin_channel = CHANNEL\_UNUSED,
    uint32_t finish_after_outputframe = 0,
    uint32_t n_frame_average = 1,
    bool pre_initialize = true )
```

construct a [FlimAbstract](#) object, [Flim](#) and [FlimBase](#) classes inherit from it

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>start_channel</i>	channel on which start clicks are received for the time differences histogramming
<i>click_channel</i>	channel on which clicks are received for the time differences histogramming
<i>pixel_begin_channel</i>	start of a pixel (histogram)
<i>n_pixels</i>	number of pixels (histograms) of one frame
<i>n_bins</i>	number of histogram bins for each pixel
<i>binwidth</i>	bin size in picoseconds
<i>pixel_end_channel</i>	end marker of a pixel - incoming clicks on the click_channel will be ignored afterwards

## Parameters

<i>frame_begin_channel</i>	(optional) start the frame, or reset the pixel index
<i>finish_after_outputframe</i>	(optional) sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0 (default value), one frame is stored and the measurement runs continuously.
<i>n_frame_average</i>	(optional) average multiple input frames into one output frame, default: 1
<i>pre_initialize</i>	(optional) initializes the measurement on constructing.

**9.26.2.2 ~FlimAbstract()**

```
FlimAbstract::~~FlimAbstract ( )
```

**9.26.3 Member Function Documentation****9.26.3.1 clear\_impl()**

```
void FlimAbstract::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.26.3.2 isAcquiring()**

```
bool FlimAbstract::isAcquiring ( ) const [inline]
```

tells if the data acquisition has finished reaching `finish_after_outputframe`

This function returns a boolean which tells the user if the class is still acquiring data. It can only reach the false state for `finish_after_outputframe > 0`.

**Note**

This can differ from `isRunning`. The return value of `isRunning` state depends only on `start/startFor/stop`.

**9.26.3.3 next\_impl()**

```
bool FlimAbstract::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.26.3.4 on\_frame\_end()**

```
virtual void FlimAbstract::on_frame_end ( ) [protected], [pure virtual]
```

Implemented in [FlimBase](#), and [Flim](#).

**9.26.3.5 on\_start()**

```
void FlimAbstract::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.26.3.6 process\_tags()**

```
template<FastBinning::Mode bin_mode>
void FlimAbstract::process_tags (
    const std::vector< Tag > & incoming_tags ) [protected]
```

**9.26.4 Member Data Documentation****9.26.4.1 acquiring**

```
bool FlimAbstract::acquiring {} [protected]
```

**9.26.4.2 acquisition\_lock**

```
std::recursive_mutex FlimAbstract::acquisition_lock [protected]
```

#### 9.26.4.3 binner

```
FastBinning FlimAbstract::binner [protected]
```

#### 9.26.4.4 binwidth

```
const timestamp_t FlimAbstract::binwidth [protected]
```

#### 9.26.4.5 click\_channel

```
const channel_t FlimAbstract::click_channel [protected]
```

#### 9.26.4.6 current\_frame\_begin

```
timestamp_t FlimAbstract::current_frame_begin [protected]
```

#### 9.26.4.7 current\_frame\_end

```
timestamp_t FlimAbstract::current_frame_end [protected]
```

#### 9.26.4.8 data\_base

```
size_t FlimAbstract::data_base {} [protected]
```

#### 9.26.4.9 finish\_after\_outputframe

```
const uint32_t FlimAbstract::finish_after_outputframe [protected]
```

#### 9.26.4.10 frame

```
std::vector<uint32_t> FlimAbstract::frame [protected]
```

#### 9.26.4.11 frame\_acquisition

```
bool FlimAbstract::frame_acquisition {} [protected]
```

#### 9.26.4.12 frame\_begin\_channel

```
const channel_t FlimAbstract::frame_begin_channel [protected]
```



#### 9.26.4.13 frames\_completed

```
uint32_t FlimAbstract::frames_completed {} [protected]
```

#### 9.26.4.14 initialized

```
bool FlimAbstract::initialized [protected]
```

#### 9.26.4.15 n\_bins

```
const uint32_t FlimAbstract::n_bins [protected]
```

#### 9.26.4.16 n\_frame\_average

```
const uint32_t FlimAbstract::n_frame_average [protected]
```

#### 9.26.4.17 n\_pixels

```
const uint32_t FlimAbstract::n_pixels [protected]
```

#### 9.26.4.18 pixel\_acquisition

```
bool FlimAbstract::pixel_acquisition {} [protected]
```

#### 9.26.4.19 pixel\_begin\_channel

```
const channel\_t FlimAbstract::pixel_begin_channel [protected]
```

#### 9.26.4.20 pixel\_begins

```
std::vector<timestamp\_t> FlimAbstract::pixel_begins [protected]
```

#### 9.26.4.21 pixel\_end\_channel

```
const channel\_t FlimAbstract::pixel_end_channel [protected]
```

#### 9.26.4.22 pixel\_ends

```
std::vector<timestamp\_t> FlimAbstract::pixel_ends [protected]
```

#### 9.26.4.23 pixels\_processed

```
uint32_t FlimAbstract::pixels_processed {} [protected]
```

#### 9.26.4.24 previous\_starts

```
std::deque<timestamp_t> FlimAbstract::previous_starts [protected]
```

#### 9.26.4.25 start\_channel

```
const channel_t FlimAbstract::start_channel [protected]
```

#### 9.26.4.26 ticks

```
uint32_t FlimAbstract::ticks {} [protected]
```

#### 9.26.4.27 time\_window

```
const timestamp_t FlimAbstract::time_window [protected]
```

The documentation for this class was generated from the following file:

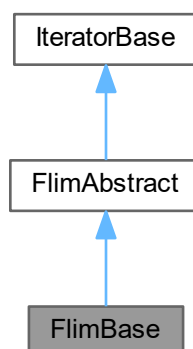
- [Iterators.h](#)

## 9.27 FlimBase Class Reference

basic measurement, containing a minimal set of features for efficiency purposes

```
#include <Iterators.h>
```

Inheritance diagram for FlimBase:



## Public Member Functions

- **FlimBase** ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, [channel\\_t](#) click\_channel, [channel\\_t](#) pixel\_begin\_channel, [uint32\\_t](#) n\_pixels, [uint32\\_t](#) n\_bins, [timestamp\\_t](#) binwidth, [channel\\_t](#) pixel\_end\_channel=CHANNEL\_UNUSED, [channel\\_t](#) frame\_begin\_channel=CHANNEL\_UNUSED, [uint32\\_t](#) finish\_after\_outputframe=0, [uint32\\_t](#) n\_frame\_average=1, [bool](#) pre\_initialize=true)  
*construct a basic [Flim](#) measurement, containing a minimum featureset for efficiency purposes*
- **~FlimBase** ()
- **void initialize** ()  
*initializes and starts measuring this [Flim](#) measurement*

## Public Member Functions inherited from [FlimAbstract](#)

- **FlimAbstract** ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, [channel\\_t](#) click\_channel, [channel\\_t](#) pixel\_begin\_channel, [uint32\\_t](#) n\_pixels, [uint32\\_t](#) n\_bins, [timestamp\\_t](#) binwidth, [channel\\_t](#) pixel\_end\_channel=CHANNEL\_UNUSED, [channel\\_t](#) frame\_begin\_channel=CHANNEL\_UNUSED, [uint32\\_t](#) finish\_after\_outputframe=0, [uint32\\_t](#) n\_frame\_average=1, [bool](#) pre\_initialize=true)  
*construct a [FlimAbstract](#) object, [Flim](#) and [FlimBase](#) classes inherit from it*
- **~FlimAbstract** ()
- **bool isAcquiring** () const  
*tells if the data acquisition has finished reaching finish\_after\_outputframe*

## Public Member Functions inherited from [IteratorBase](#)

- **virtual ~IteratorBase** ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- **void start** ()  
*Starts or continues data acquisition.*
- **void startFor** ([timestamp\\_t](#) capture\_duration, [bool](#) clear=true)  
*Starts or continues the data acquisition for the given duration.*
- **bool waitUntilFinished** ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- **void stop** ()  
*After calling this method, the measurement will stop processing incoming tags.*
- **void clear** ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- **void abort** ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- **bool isRunning** ()  
*Returns True if the measurement is collecting the data.*
- **[timestamp\\_t](#) getCaptureDuration** ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- **[std::string](#) getConfiguration** ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- **void on\_frame\_end** () override final
- **virtual void frameReady** ([uint32\\_t](#) frame\_number, [std::vector](#)< [uint32\\_t](#) > &data, [std::vector](#)< [timestamp\\_t](#) > &pixel\_begin\_times, [std::vector](#)< [timestamp\\_t](#) > &pixel\_end\_times, [timestamp\\_t](#) frame\_begin\_time, [timestamp\\_t](#) frame\_end\_time)

## Protected Member Functions inherited from [FlimAbstract](#)

- `template<FastBinning::Mode bin_mode>`  
`void process\_tags (const std::vector< Tag > &incoming_tags)`
- `bool next\_impl (std::vector< Tag > &incoming_tags, timestamp\_t begin_time, timestamp\_t end_time) override`  
*update iterator state*
- `void clear\_impl () override`  
*clear [Iterator](#) state.*
- `void on\_start () override`  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- `IteratorBase (TimeTaggerBase *tagger, std::string base_type_="IteratorBase", std::string extra_info_="")`  
*Standard constructor, which will register with the Time Tagger backend.*
- `void registerChannel (channel\_t channel)`  
*register a channel*
- `void unregisterChannel (channel\_t channel)`  
*unregister a channel*
- `channel\_t getNewVirtualChannel ()`  
*allocate a new virtual output channel for this iterator*
- `void finishInitialization ()`  
*method to call after finishing the initialization of the measurement*
- `virtual void on\_stop ()`  
*callback when the measurement class is stopped*
- `void lock ()`  
*acquire update lock*
- `void unlock ()`  
*release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`  
*acquire update lock*
- `void finish\_running ()`  
*Callback for the measurement to stop itself.*
- `void checkForAbort ()`
- `template<typename T >`  
`void checkForAbort (T callback)`

## Protected Attributes

- `uint32_t total\_frames`

## Protected Attributes inherited from [FlimAbstract](#)

- const [channel\\_t](#) [start\\_channel](#)
- const [channel\\_t](#) [click\\_channel](#)
- const [channel\\_t](#) [pixel\\_begin\\_channel](#)
- const [uint32\\_t](#) [n\\_pixels](#)
- const [uint32\\_t](#) [n\\_bins](#)
- const [timestamp\\_t](#) [binwidth](#)
- const [channel\\_t](#) [pixel\\_end\\_channel](#)
- const [channel\\_t](#) [frame\\_begin\\_channel](#)
- const [uint32\\_t](#) [finish\\_after\\_outputframe](#)
- const [uint32\\_t](#) [n\\_frame\\_average](#)
- const [timestamp\\_t](#) [time\\_window](#)
- [timestamp\\_t](#) [current\\_frame\\_begin](#)
- [timestamp\\_t](#) [current\\_frame\\_end](#)
- bool [acquiring](#) {}
- bool [frame\\_acquisition](#) {}
- bool [pixel\\_acquisition](#) {}
- [uint32\\_t](#) [pixels\\_processed](#) {}
- [uint32\\_t](#) [frames\\_completed](#) {}
- [uint32\\_t](#) [ticks](#) {}
- [size\\_t](#) [data\\_base](#) {}
- [std::vector< uint32\\_t >](#) [frame](#)
- [std::vector< timestamp\\_t >](#) [pixel\\_begins](#)
- [std::vector< timestamp\\_t >](#) [pixel\\_ends](#)
- [std::deque< timestamp\\_t >](#) [previous\\_starts](#)
- [FastBinning](#) [binner](#)
- [std::recursive\\_mutex](#) [acquisition\\_lock](#)
- bool [initialized](#)

## Protected Attributes inherited from [IteratorBase](#)

- [std::set< channel\\_t >](#) [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- [std::atomic< bool >](#) [aborting](#)

### 9.27.1 Detailed Description

basic measurement, containing a minimal set of features for efficiency purposes

The [FlimBase](#) provides only the most essential functionality for FLIM tasks. The benefit from the reduced functionality is that it is very memory and CPU efficient. The class provides the [frameReady\(\)](#) callback, which must be used to analyze the data.

## 9.27.2 Constructor & Destructor Documentation

### 9.27.2.1 FlimBase()

```

FlimBase::FlimBase (
    TimeTaggerBase * tagger,
    channel_t start_channel,
    channel_t click_channel,
    channel_t pixel_begin_channel,
    uint32_t n_pixels,
    uint32_t n_bins,
    timestamp_t binwidth,
    channel_t pixel_end_channel = CHANNEL_UNUSED,
    channel_t frame_begin_channel = CHANNEL_UNUSED,
    uint32_t finish_after_outputframe = 0,
    uint32_t n_frame_average = 1,
    bool pre_initialize = true )

```

construct a basic [Flim](#) measurement, containing a minimum featureset for efficiency purposes

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>start_channel</i>	channel on which start clicks are received for the time differences histogramming
<i>click_channel</i>	channel on which clicks are received for the time differences histogramming
<i>pixel_begin_channel</i>	start of a pixel (histogram)
<i>n_pixels</i>	number of pixels (histograms) of one frame
<i>n_bins</i>	number of histogram bins for each pixel
<i>binwidth</i>	bin size in picoseconds
<i>pixel_end_channel</i>	end marker of a pixel - incoming clicks on the click_channel will be ignored afterwards
<i>frame_begin_channel</i>	(optional) start the frame, or reset the pixel index
<i>finish_after_outputframe</i>	(optional) sets the number of frames stored within the measurement class. After reaching the number, the measurement will stop. If the number is 0 (default value), one frame is stored and the measurement runs continuously.
<i>n_frame_average</i>	(optional) average multiple input frames into one output frame, default: 1
<i>pre_initialize</i>	(optional) initializes the measurement on constructing.

### 9.27.2.2 ~FlimBase()

```

FlimBase::~FlimBase ( )

```

## 9.27.3 Member Function Documentation

### 9.27.3.1 frameReady()

```

virtual void FlimBase::frameReady (
    uint32_t frame_number,
    std::vector< uint32_t > & data,

```

```
std::vector< timestamp_t > & pixel_begin_times,
std::vector< timestamp_t > & pixel_end_times,
timestamp_t frame_begin_time,
timestamp_t frame_end_time ) [protected], [virtual]
```

### 9.27.3.2 initialize()

```
void FlimBase::initialize ( )
```

initializes and starts measuring this [Flim](#) measurement

This function initializes the [Flim](#) measurement and starts executing it. It does nothing if preinitialized in the constructor is set to true.

### 9.27.3.3 on\_frame\_end()

```
void FlimBase::on_frame_end ( ) [final], [override], [protected], [virtual]
```

Implements [FlimAbstract](#).

## 9.27.4 Member Data Documentation

### 9.27.4.1 total\_frames

```
uint32_t FlimBase::total_frames [protected]
```

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.28 FlimFrameInfo Class Reference

object for storing the state of [Flim::getCurrentFrameEx](#)

```
#include <Iterators.h>
```

### Public Member Functions

- [~FlimFrameInfo](#) ()
- [int32\\_t getFrameNumber](#) () const  
*index of this frame*
- [bool isValid](#) () const  
*tells if this frame is valid*
- [uint32\\_t getPixelPosition](#) () const  
*number of pixels acquired on this frame*
- [void getHistograms](#) (std::function< uint32\_t \*(size\_t, size\_t)> array\_out)
- [void getIntensities](#) (std::function< float \*(size\_t)> array\_out)
- [void getSummedCounts](#) (std::function< uint64\_t \*(size\_t)> array\_out)
- [void getPixelBegins](#) (std::function< timestamp\_t \*(size\_t)> array\_out)
- [void getPixelEnds](#) (std::function< timestamp\_t \*(size\_t)> array\_out)

## Public Attributes

- uint32\_t [pixels](#)
- uint32\_t [bins](#)
- int32\_t [frame\\_number](#)
- uint32\_t [pixel\\_position](#)
- bool [valid](#)

## 9.28.1 Detailed Description

object for storing the state of [Flim::getCurrentFrameEx](#)

## 9.28.2 Constructor & Destructor Documentation

### 9.28.2.1 ~FlimFrameInfo()

```
FlimFrameInfo::~FlimFrameInfo ( )
```

## 9.28.3 Member Function Documentation

### 9.28.3.1 getFrameNumber()

```
int32_t FlimFrameInfo::getFrameNumber ( ) const [inline]
```

index of this frame

This function 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

deprecated, use `frame_number` instead..

### 9.28.3.2 getHistograms()

```
void FlimFrameInfo::getHistograms (
    std::function< uint32_t *(size_t, size_t)> array_out )
```

### 9.28.3.3 getIntensities()

```
void FlimFrameInfo::getIntensities (
    std::function< float *(size_t)> array_out )
```

### 9.28.3.4 getPixelBegins()

```
void FlimFrameInfo::getPixelBegins (
    std::function< timestamp_t *(size_t)> array_out )
```



### 9.28.3.5 getPixelEnds()

```
void FlimFrameInfo::getPixelEnds (
    std::function< timestamp_t *(size_t)> array_out )
```

### 9.28.3.6 getPixelPosition()

```
uint32_t FlimFrameInfo::getPixelPosition ( ) const [inline]
```

number of pixels acquired on this frame

This function returns a value which tells how many pixels were processed for this frame.

### 9.28.3.7 getSummedCounts()

```
void FlimFrameInfo::getSummedCounts (
    std::function< uint64_t *(size_t)> array_out )
```

### 9.28.3.8 isValid()

```
bool FlimFrameInfo::isValid ( ) const [inline]
```

tells if this frame is valid

This function returns a boolean which tells if this frame is valid or not. Invalid frames are possible on errors, such as asking for the last completed frame when no frame has been completed so far.

deprecated, use isValid instead.

## 9.28.4 Member Data Documentation

### 9.28.4.1 bins

```
uint32_t FlimFrameInfo::bins
```

### 9.28.4.2 frame\_number

```
int32_t FlimFrameInfo::frame_number
```

### 9.28.4.3 pixel\_position

```
uint32_t FlimFrameInfo::pixel_position
```

#### 9.28.4.4 pixels

```
uint32_t FlimFrameInfo::pixels
```

#### 9.28.4.5 valid

```
bool FlimFrameInfo::valid
```

The documentation for this class was generated from the following file:

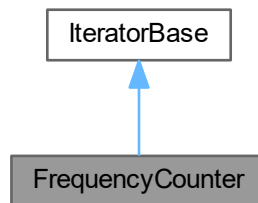
- [Iterators.h](#)

## 9.29 FrequencyCounter Class Reference

Calculate the phase of multiple channels at equidistant sampling points.

```
#include <Iterators.h>
```

Inheritance diagram for FrequencyCounter:



### Public Member Functions

- [FrequencyCounter](#) ([TimeTaggerBase](#) \*tagger, std::vector< [channel\\_t](#) > channels, [timestamp\\_t](#) sampling\_↔ interval, [timestamp\\_t](#) fitting\_window, int32\_t n\_values=0)
- [~FrequencyCounter](#) ()
- [FrequencyCounterData](#) [getDataObject](#) (uint16\_t event\_divider=1, bool remove=false, bool channels\_last\_↔ dim=false)

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- [std::unique\\_lock< std::mutex > getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- [template<typename T > void checkForAbort](#) (T callback)

### Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- [std::set< channel\\_t > channels\\_registered](#)
- list of channels used by the iterator*
- bool [running](#)
- running state of the iterator*
- bool [autostart](#)
- Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase \\* tagger](#)
- Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t capture\\_duration](#)
- Duration the iterator has already processed data.*
- [timestamp\\_t pre\\_capture\\_duration](#)
- For internal use.*
- [std::atomic< bool > aborting](#)

## 9.29.1 Detailed Description

Calculate the phase of multiple channels at equidistant sampling points.

This measurement calculates the phase of a periodic signal at evenly spaced sampling times. 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.

## 9.29.2 Constructor & Destructor Documentation

### 9.29.2.1 FrequencyCounter()

```
FrequencyCounter::FrequencyCounter (
    TimeTaggerBase * tagger,
    std::vector< channel_t > channels,
    timestamp_t sampling_interval,
    timestamp_t fitting_window,
    int32_t n_values = 0 )
```

### 9.29.2.2 ~FrequencyCounter()

```
FrequencyCounter::~~FrequencyCounter ( )
```

## 9.29.3 Member Function Documentation

### 9.29.3.1 clear\_impl()

```
void FrequencyCounter::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.29.3.2 getDataObject()

```
FrequencyCounterData FrequencyCounter::getDataObject (
    uint16_t event_divider = 1,
    bool remove = false,
    bool channels_last_dim = false )
```

### 9.29.3.3 next\_impl()

```
bool FrequencyCounter::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.29.3.4 on\_start()

```
void FrequencyCounter::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.30 FrequencyCounterData Class Reference

```
#include <Iterators.h>
```

### Public Member Functions

- [~FrequencyCounterData](#) ()
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)  
*Returns the index of each sampling point.*
- void [getTime](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)  
*get the timestamp of the bins since the last clear*
- void [getOverflowMask](#) (std::function< signed char \*(size\_t, size\_t)> array\_out)  
*Returns an overflow mask with 1 = "has overflow" and 0 = "is valid".*
- void [getPeriodsCount](#) (std::function< [timestamp\\_t](#) \*(size\_t, size\_t)> array\_out)  
*Integer part of the absolute phase.*
- void [getPeriodsFraction](#) (std::function< double \*(size\_t, size\_t)> array\_out)  
*Fraction of the current cycle as a value from [0, 1).*
- void [getFrequency](#) (std::function< double \*(size\_t, size\_t)> array\_out, [timestamp\\_t](#) time\_scale=1000000000000)  
*Frequency of the previous sampling period calculated from the phase difference between the current and the previous sample.*
- void [getFrequencyInstantaneous](#) (std::function< double \*(size\_t, size\_t)> array\_out)  
*Instantaneous frequency within the fitting window obtained from the fit slope.*
- void [getPhase](#) (std::function< double \*(size\_t, size\_t)> array\_out, double reference\_frequency=0)  
*Phase with respect to an expected reference frequency.*

### Public Attributes

- const [timestamp\\_t](#) [overflow\\_samples](#)  
*Number of sampling points affected by an overflow range since the start of the measurement.*
- const unsigned int [size](#)  
*Number of sampling points represented by the object.*
- const bool [align\\_to\\_reference](#)  
*Indicates if the sampling grid has been aligned to the SoftwareClock.*
- const [timestamp\\_t](#) [sampling\\_interval](#)  
*The sampling interval in picoseconds.*
- const [timestamp\\_t](#) [sample\\_offset](#)  
*Index offset of the first index.*
- const bool [channels\\_last\\_dim](#)  
*Channels as last dimension.*

## 9.30.1 Constructor & Destructor Documentation

### 9.30.1.1 ~FrequencyCounterData()

```
FrequencyCounterData::~FrequencyCounterData ( )
```

## 9.30.2 Member Function Documentation

### 9.30.2.1 getFrequency()

```
void FrequencyCounterData::getFrequency (
    std::function< double *(size_t, size_t)> array_out,
    timestamp_t time_scale = 1000000000000 )
```

Frequency of the previous sampling period calculated from the phase difference between the current and the previous sample.

### 9.30.2.2 getFrequencyInstantaneous()

```
void FrequencyCounterData::getFrequencyInstantaneous (
    std::function< double *(size_t, size_t)> array_out )
```

Instantaneous frequency within the fitting window obtained from the fit slope.

### 9.30.2.3 getIndex()

```
void FrequencyCounterData::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

Returns the index of each sampling point.

### 9.30.2.4 getOverflowMask()

```
void FrequencyCounterData::getOverflowMask (
    std::function< signed char *(size_t, size_t)> array_out )
```

Returns an overflow mask with 1 = "has overflow" and 0 = "is valid".

### 9.30.2.5 getPeriodsCount()

```
void FrequencyCounterData::getPeriodsCount (
    std::function< timestamp_t *(size_t, size_t)> array_out )
```

Integer part of the absolute phase.

### 9.30.2.6 getPeriodsFraction()

```
void FrequencyCounterData::getPeriodsFraction (
    std::function< double *(size_t, size_t)> array_out )
```

Fraction of the current cycle as a value from [0, 1).

### 9.30.2.7 getPhase()

```
void FrequencyCounterData::getPhase (
    std::function< double *(size_t, size_t)> array_out,
    double reference_frequency = 0 )
```

Phase with respect to an expected reference frequency.

### 9.30.2.8 getTime()

```
void FrequencyCounterData::getTime (
    std::function< timestamp_t *(size_t)> array_out )
```

get the timestamp of the bins since the last clear

## 9.30.3 Member Data Documentation

### 9.30.3.1 align\_to\_reference

```
const bool FrequencyCounterData::align_to_reference
```

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

### 9.30.3.2 channels\_last\_dim

```
const bool FrequencyCounterData::channels_last_dim
```

Channels as last dimension.

### 9.30.3.3 overflow\_samples

```
const timestamp_t FrequencyCounterData::overflow_samples
```

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

### 9.30.3.4 sample\_offset

```
const timestamp_t FrequencyCounterData::sample_offset
```

Index offset of the first index.



### 9.30.3.5 sampling\_interval

```
const timestamp_t FrequencyCounterData::sampling_interval
```

The sampling interval in picoseconds.

### 9.30.3.6 size

```
const unsigned int FrequencyCounterData::size
```

Number of sampling points represented by the object.

The documentation for this class was generated from the following file:

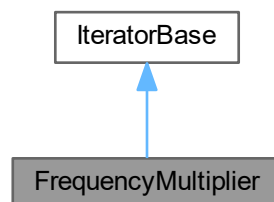
- [Iterators.h](#)

## 9.31 FrequencyMultiplier Class Reference

The signal of an input channel is scaled up to a higher frequency according to the multiplier passed as a parameter.

```
#include <Iterators.h>
```

Inheritance diagram for FrequencyMultiplier:



### Public Member Functions

- [FrequencyMultiplier](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, [int32\\_t](#) multiplier)  
*constructor of a [FrequencyMultiplier](#)*
- [~FrequencyMultiplier](#) ()
- [channel\\_t](#) [getChannel](#) ()
- [int32\\_t](#) [getMultiplier](#) ()

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`  
void `checkForAbort (T callback)`

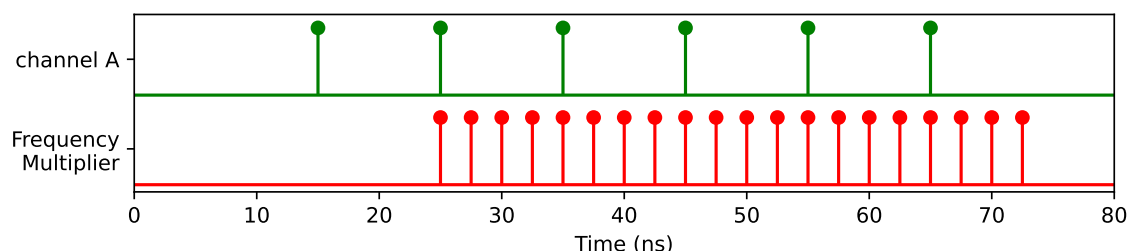
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.31.1 Detailed Description

The signal of an input channel is scaled up to a higher frequency according to the multiplier passed as a parameter.



The `FrequencyMultiplier` inserts copies 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 linearly the intermediate timestamps to match the upscaled 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. For example a 80 MHz laser sync signal can be scaled down via `setEventDivider(..., 80)` to 1 MHz (hardware side) and an 80 MHz signal can be restored via `FrequencyMultiplier(..., 80)` on the software side with some loss in precision. The `FrequencyMultiplier` is an alternative way to reduce the data rate in comparison to the `EventFilter`, which has a higher precision but can be more difficult to use.

## 9.31.2 Constructor & Destructor Documentation

### 9.31.2.1 FrequencyMultiplier()

```
FrequencyMultiplier::FrequencyMultiplier (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    int32_t multiplier )
```

constructor of a [FrequencyMultiplier](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel on which the upscaling of the frequency is based on
<i>multiplier</i>	frequency upscaling factor

### 9.31.2.2 ~FrequencyMultiplier()

```
FrequencyMultiplier::~~FrequencyMultiplier ( )
```

## 9.31.3 Member Function Documentation

### 9.31.3.1 clear\_impl()

```
void FrequencyMultiplier::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.31.3.2 getChannel()

```
channel_t FrequencyMultiplier::getChannel ( )
```

### 9.31.3.3 getMultiplier()

```
int32_t FrequencyMultiplier::getMultiplier ( )
```

### 9.31.3.4 next\_impl()

```
bool FrequencyMultiplier::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

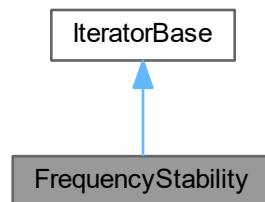
- [Iterators.h](#)

## 9.32 FrequencyStability Class Reference

Allan deviation (and related metrics) calculator.

```
#include <Iterators.h>
```

Inheritance diagram for FrequencyStability:



## Public Member Functions

- [FrequencyStability](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) channel, std::vector< [uint64\\_t](#) > steps, [timestamp\\_t](#) average=1000, [uint64\\_t](#) trace\_len=1000)  
*constructor of a [FrequencyStability](#) measurement*
- [~FrequencyStability](#) ()
- [FrequencyStabilityData](#) [getDataObject](#) ()  
*get a return object with all data in a synchronized way*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock` ()
- release update lock*
- `OrderedBarrier::OrderInstance parallelize` (`OrderedPipeline` &pipeline)
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock` ()
- acquire update lock*
- void `finish_running` ()
- Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

#### 9.32.1 Detailed Description

Allan deviation (and related metrics) calculator.

It shall analyze the stability of a clock by computing deviations of  $\text{phase}[i] - \text{phase}[i + n]$ . The list of all  $n$  values needs to be declared in the beginning.

Reference: <https://www.nist.gov/publications/handbook-frequency-stability-analysis>

It calculates the STDD, ADEV, MDEV and HDEV on the fly:

- STDD: Standard derivation of each period pair. This is not a stable analysis with frequency drifts and only calculated for reference.
- ADEV: Overlapping Allan deviation, the most common analysis framework. Square mean value of the second derivate  $\text{phase}[i] - 2*\text{phase}[i + n] + \text{phase}[i + 2*n]$ . In a loglog plot, the slope allows to identify the source 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. thermal

As this tool is most likely used to analyze timings, a scaled ADEV is implemented. It adds 1.0 to each slope and normalize the return value to picoseconds for phase noise.

- MDEV: Modified overlapping Allan deviation. It averages the second derivate of ADEV before calculating the MSE. 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 scaled approach (+1 on each slope yielding picoseconds as return value) is called TDEV and more commonly used than MDEV.

- HDEV: The overlapping Hadamard deviation uses the third derivate of the phase. This cancels the effect of a constant phase drift.

## 9.32.2 Constructor & Destructor Documentation

### 9.32.2.1 FrequencyStability()

```
FrequencyStability::FrequencyStability (
    TimeTaggerBase * tagger,
    channel_t channel,
    std::vector< uint64_t > steps,
    timestamp_t average = 1000,
    uint64_t trace_len = 1000 )
```

constructor of a [FrequencyStability](#) measurement

#### Parameters

<i>tagger</i>	time tagger object
<i>channel</i>	the clock input channel used for the analysis
<i>steps</i>	a vector or integer tau values for all deviations
<i>average</i>	an averaging down sampler to reduce noise and memory requirements
<i>trace_len</i>	length of the phase and frequency trace capture of the averaged data

#### Note

This measurements needs 24 times the largest value in steps bytes of main memory

### 9.32.2.2 ~FrequencyStability()

```
FrequencyStability::~FrequencyStability ( )
```



### 9.32.3 Member Function Documentation

#### 9.32.3.1 clear\_impl()

```
void FrequencyStability::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.32.3.2 getDataObject()

```
FrequencyStabilityData FrequencyStability::getDataObject ( )
```

get a return object with all data in a synchronized way

#### 9.32.3.3 next\_impl()

```
bool FrequencyStability::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

##### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

##### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

#### 9.32.3.4 on\_start()

```
void FrequencyStability::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.33 FrequencyStabilityData Class Reference

return data object for FrequencyStability::getData.

```
#include <Iterators.h>
```

### Public Member Functions

- [~FrequencyStabilityData](#) ()
- void [getSTDD](#) (std::function< double \*(size\_t)> array\_out)  
*returns the standard derivation of each period pair*
- void [getADEV](#) (std::function< double \*(size\_t)> array\_out)  
*returns the overlapping Allan deviation*
- void [getMDEV](#) (std::function< double \*(size\_t)> array\_out)  
*returns the modified overlapping Allan deviation*
- void [getTDEV](#) (std::function< double \*(size\_t)> array\_out)  
*returns the overlapping time deviation*
- void [getHDEV](#) (std::function< double \*(size\_t)> array\_out)  
*returns the overlapping Hadamard deviation*
- void [getADEVScaled](#) (std::function< double \*(size\_t)> array\_out)  
*returns the scaled version of the overlapping Allan deviation*
- void [getHDEVScaled](#) (std::function< double \*(size\_t)> array\_out)  
*returns the scaled version of the overlapping Hadamard deviation*
- void [getTau](#) (std::function< double \*(size\_t)> array\_out)  
*returns the analysis position of all deviations*
- void [getTracePhase](#) (std::function< double \*(size\_t)> array\_out)  
*returns a trace of the last phase samples in seconds*
- void [getTraceFrequency](#) (std::function< double \*(size\_t)> array\_out)  
*returns a trace of the last normalized frequency error samples in pp1*
- void [getTraceFrequencyAbsolute](#) (std::function< double \*(size\_t)> array\_out, double input\_frequency=0.0)  
*returns a trace of the last absolute frequency samples in Hz*
- void [getTraceIndex](#) (std::function< double \*(size\_t)> array\_out)  
*returns the timestamps of the traces in seconds*

### 9.33.1 Detailed Description

return data object for FrequencyStability::getData.

## 9.33.2 Constructor & Destructor Documentation

### 9.33.2.1 ~FrequencyStabilityData()

FrequencyStabilityData::~~FrequencyStabilityData ( )

## 9.33.3 Member Function Documentation

### 9.33.3.1 getADEV()

```
void FrequencyStabilityData::getADEV (
    std::function< double *(size_t)> array_out )
```

returns the overlapping Allan deviation

### 9.33.3.2 getADEVScaled()

```
void FrequencyStabilityData::getADEVScaled (
    std::function< double *(size_t)> array_out )
```

returns the scaled version of the overlapping Allan deviation

### 9.33.3.3 getHDEV()

```
void FrequencyStabilityData::getHDEV (
    std::function< double *(size_t)> array_out )
```

returns the overlapping Hadamard deviation

### 9.33.3.4 getHDEVScaled()

```
void FrequencyStabilityData::getHDEVScaled (
    std::function< double *(size_t)> array_out )
```

returns the scaled version of the overlapping Hadamard deviation

### 9.33.3.5 getMDEV()

```
void FrequencyStabilityData::getMDEV (
    std::function< double *(size_t)> array_out )
```

returns the modified overlapping Allan deviation

**9.33.3.6 getSTDD()**

```
void FrequencyStabilityData::getSTDD (
    std::function< double *(size_t)> array_out )
```

returns the standard derivation of each period pair

**9.33.3.7 getTau()**

```
void FrequencyStabilityData::getTau (
    std::function< double *(size_t)> array_out )
```

returns the analysis position of all deviations

**9.33.3.8 getTDEV()**

```
void FrequencyStabilityData::getTDEV (
    std::function< double *(size_t)> array_out )
```

returns the overlapping time deviation

This is the scaled version of the modified overlapping Allan deviation.

**9.33.3.9 getTraceFrequency()**

```
void FrequencyStabilityData::getTraceFrequency (
    std::function< double *(size_t)> array_out )
```

returns a trace of the last normalized frequency error samples in pp1

**9.33.3.10 getTraceFrequencyAbsolute()**

```
void FrequencyStabilityData::getTraceFrequencyAbsolute (
    std::function< double *(size_t)> array_out,
    double input_frequency = 0.0 )
```

returns a trace of the last absolute frequency samples in Hz

**Parameters**

<i>array_out</i>	allocator for return array
<i>input_frequency</i>	reference frequency in Hz

**Note**

The precision of the parameter `input_frequency` and so the mean value of the return values are limited to 15 digits. However the relative errors within the return values have a higher precision.

## 9.33.3.11 getTraceIndex()

```
void FrequencyStabilityData::getTraceIndex (
    std::function< double *(size_t)> array_out )
```

returns the timestamps of the traces in seconds

## 9.33.3.12 getTracePhase()

```
void FrequencyStabilityData::getTracePhase (
    std::function< double *(size_t)> array_out )
```

returns a trace of the last phase samples in seconds

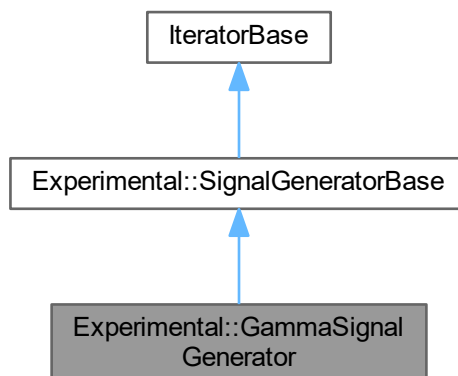
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.34 Experimental::GammaSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::GammaSignalGenerator:



## Public Member Functions

- [GammaSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double alpha, double beta, [channel\\_t](#) base\_↔ channel=[CHANNEL\\_UNUSED](#), int32\_t seed=-1)  
Construct a gamma event channel.
- [~GammaSignalGenerator](#) ()

## Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- void [initialize](#) ([timestamp\\_t](#) [initial\\_time](#)) override
- [timestamp\\_t](#) [get\\_next](#) () override
- void [on\\_restart](#) ([timestamp\\_t](#) [restart\\_time](#)) override

## Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#)) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool [is\\_finished](#))

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- std::unique\_ptr< [SignalGeneratorBaseImpl](#) > impl

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

## 9.34.1 Constructor & Destructor Documentation

### 9.34.1.1 GammaSignalGenerator()

```
Experimental::GammaSignalGenerator::GammaSignalGenerator (
    TimeTaggerBase * tagger,
    double alpha,
    double beta,
    channel_t base_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

Construct a gamma event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>alpha</i>	alpha value of the gamma distribution
<i>beta</i>	beta value of the gamma distribution
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.34.1.2 ~GammaSignalGenerator()

```
Experimental::GammaSignalGenerator::~~GammaSignalGenerator ( )
```

## 9.34.2 Member Function Documentation

### 9.34.2.1 get\_next()

```
timestamp_t Experimental::GammaSignalGenerator::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.34.2.2 initialize()

```
void Experimental::GammaSignalGenerator::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.34.2.3 on\_restart()

```
void Experimental::GammaSignalGenerator::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

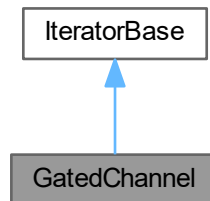


## 9.35 GatedChannel Class Reference

An input channel is gated by a gate channel.

```
#include <Iterators.h>
```

Inheritance diagram for GatedChannel:



### Public Member Functions

- `GatedChannel` (`TimeTaggerBase` \*tagger, `channel_t` input\_channel, `channel_t` gate\_start\_channel, `channel_t` gate\_stop\_channel, `GatedChannelInitial` initial=`GatedChannelInitial::Closed`)  
*constructor of a `GatedChannel`*
- `~GatedChannel` ()
- `channel_t` getChannel ()  
*the new virtual channel*

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void `start` ()  
*Starts or continues data acquisition.*
- void `startFor` (`timestamp_t` capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool `waitUntilFinished` (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- void `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` getCaptureDuration ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- std::string getConfiguration ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

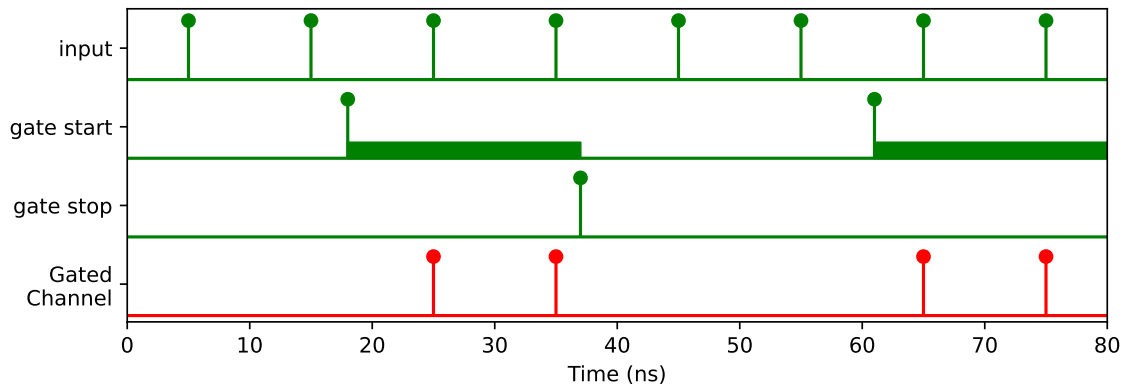
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

### 9.35.1 Detailed Description

An input channel is gated by a gate channel.



Note: The gate is edge sensitive and not level sensitive. That means that the gate will transfer data only when an appropriate level change is detected on the `gate_start_channel`.

### 9.35.2 Constructor & Destructor Documentation

#### 9.35.2.1 GatedChannel()

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

constructor of a [GatedChannel](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel which is gated
<i>gate_start_channel</i>	channel on which a signal detected will start the transmission of the <code>input_channel</code> through the gate
<i>gate_stop_channel</i>	channel on which a signal detected will stop the transmission of the <code>input_channel</code> through the gate
<i>initial</i>	initial state of the gate

#### 9.35.2.2 ~GatedChannel()

```
GatedChannel::~GatedChannel ( )
```

### 9.35.3 Member Function Documentation

#### 9.35.3.1 `clear_impl()`

```
void GatedChannel::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the `clear_impl()` method to reset its internal state. The `clear_impl()` function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.35.3.2 `getChannel()`

```
channel_t GatedChannel::getChannel ( )
```

the new virtual channel

This function returns the new allocated virtual channel. It can be used now in any new iterator.

#### 9.35.3.3 `next_impl()`

```
bool GatedChannel::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

##### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

##### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

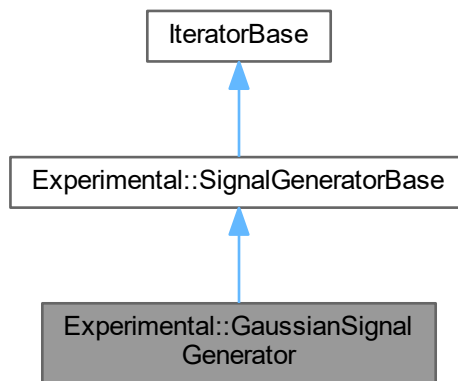
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.36 Experimental::GaussianSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::GaussianSignalGenerator:



### Public Member Functions

- [GaussianSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double mean, double standard\_deviation, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#), [int32\\_t](#) seed=-1)  
*Construct a gaussian event channel.*
- [~GaussianSignalGenerator](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
*the new virtual channel*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()

*After calling this method, the measurement will stop processing incoming tags.*

- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- void `initialize` (`timestamp_t` initial\_time) override
- `timestamp_t` `get_next` () override
- void `on_restart` (`timestamp_t` restart\_time) override

### Protected Member Functions inherited from `Experimental::SignalGeneratorBase`

- bool `next_impl` (`std::vector< Tag >` &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `on_stop` () override  
*callback when the measurement class is stopped*
- bool `isProcessingFinished` ()
- void `set_processing_finished` (bool is\_finished)

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*

- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`  
*acquire update lock*
- `void finish\_running ()`  
*Callback for the measurement to stop itself.*
- `void checkForAbort ()`
- `template<typename T >`  
`void checkForAbort (T callback)`

### Additional Inherited Members

### Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- `std::unique_ptr< SignalGeneratorBaseImpl > impl`

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.36.1 Constructor & Destructor Documentation

### 9.36.1.1 [GaussianSignalGenerator](#)()

```
Experimental::GaussianSignalGenerator::GaussianSignalGenerator (
    TimeTaggerBase * tagger,
    double mean,
    double standard\_deviation,
    channel\_t base\_channel = CHANNEL\_UNUSED,
    int32_t seed = -1 )
```

Construct a gaussian event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>mean</i>	mean time each event is generated.
<i>standard_deviation</i>	standard deviation of the normal distribution.
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.36.1.2 ~GaussianSignalGenerator()

Experimental::GaussianSignalGenerator::~~GaussianSignalGenerator ( )

## 9.36.2 Member Function Documentation

### 9.36.2.1 get\_next()

`timestamp_t` Experimental::GaussianSignalGenerator::get\_next ( ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

### 9.36.2.2 initialize()

void Experimental::GaussianSignalGenerator::initialize (   
 `timestamp_t` *initial\_time* ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

### 9.36.2.3 on\_restart()

void Experimental::GaussianSignalGenerator::on\_restart (   
 `timestamp_t` *restart\_time* ) [override], [protected], [virtual]

Reimplemented from [Experimental::SignalGeneratorBase](#).

The documentation for this class was generated from the following file:

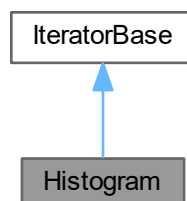
- [Iterators.h](#)

## 9.37 Histogram Class Reference

Accumulate time differences into a histogram.

```
#include <Iterators.h>
```

Inheritance diagram for Histogram:





## Public Member Functions

- [Histogram](#) ([TimeTaggerBase](#) \*[tagger](#), [channel\\_t](#) [click\\_channel](#), [channel\\_t](#) [start\\_channel](#)=[CHANNEL\\_UNUSED](#), [timestamp\\_t](#) [binwidth](#)=1000, [int32\\_t](#) [n\\_bins](#)=1000)  
*constructor of a [Histogram](#) measurement*
- [~Histogram](#) ()
- void [getData](#) (std::function< [int32\\_t](#) \*([size\\_t](#))> [array\\_out](#))
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#))> [array\\_out](#))

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &[incoming\\_tags](#), [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#)) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

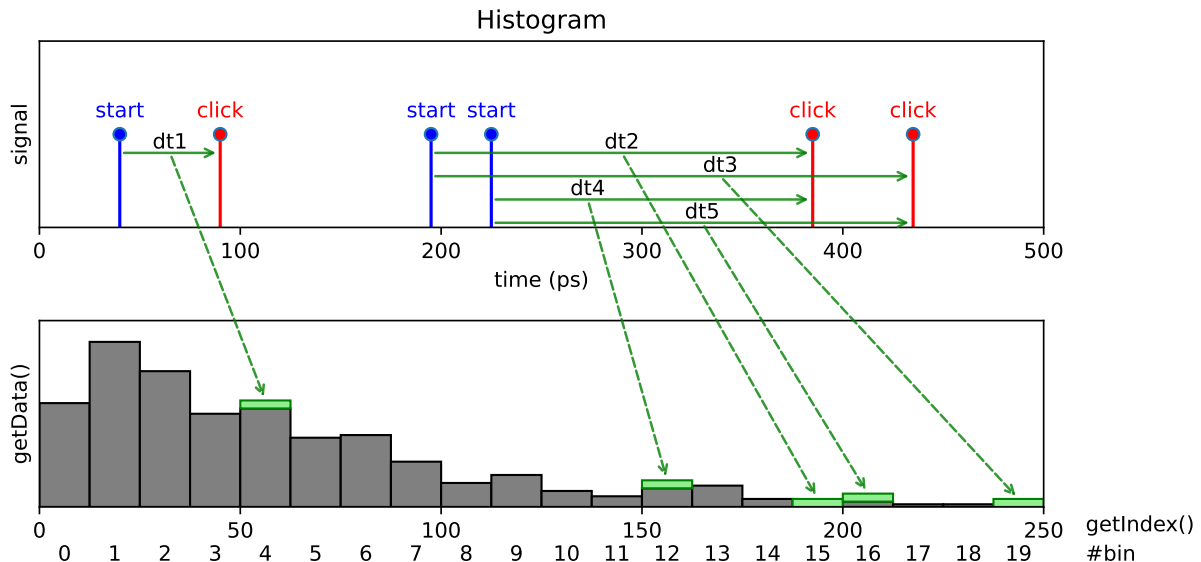
## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.37.1 Detailed Description

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 thread waits for clicks on a first channel, the 'start channel', then measures the time difference between the last start click and all subsequent clicks on a second channel, the 'click channel', and stores them in a histogram. The histogram range and resolution is specified by the number of bins and the binwidth. 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 'multiple start, multiple stop' measurement and corresponds to a full auto- or cross-correlation measurement.

### 9.37.2 Constructor & Destructor Documentation

#### 9.37.2.1 Histogram()

```
Histogram::Histogram (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel = CHANNEL_UNUSED,
    timestamp_t binwidth = 1000,
    int32_t n_bins = 1000 )
```

constructor of a [Histogram](#) measurement

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel that increments the count in a bin
<i>start_channel</i>	channel that sets start times relative to which clicks on the click channel are measured
<i>binwidth</i>	width of one histogram bin in ps
<i>n_bins</i>	number of bins in the histogram

### 9.37.2.2 ~Histogram()

```
Histogram::~Histogram ( )
```

## 9.37.3 Member Function Documentation

### 9.37.3.1 clear\_impl()

```
void Histogram::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.37.3.2 getData()

```
void Histogram::getData (
    std::function< int32_t *(size_t)> array_out )
```

### 9.37.3.3 getIndex()

```
void Histogram::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

### 9.37.3.4 next\_impl()

```
bool Histogram::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.37.3.5 on\_start()**

```
void Histogram::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

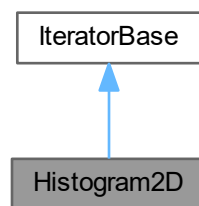
- [Iterators.h](#)

## 9.38 Histogram2D Class Reference

A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.

```
#include <Iterators.h>
```

Inheritance diagram for Histogram2D:

**Public Member 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, [int32\\_t](#) n\_bins\_1, [int32\\_t](#) n\_bins\_2)  
*constructor of a [Histogram2D](#) measurement*
- [~Histogram2D](#) ()
- void [getData](#) (std::function< [int32\\_t](#) \*([size\\_t](#), [size\\_t](#))> array\_out)
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#), [size\\_t](#), [size\\_t](#))> array\_out)
- void [getIndex\\_1](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#))> array\_out)
- void [getIndex\\_2](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#))> array\_out)

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock ()`
- release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`
- acquire update lock*
- void `finish_running ()`
- Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`  
void `checkForAbort (T callback)`

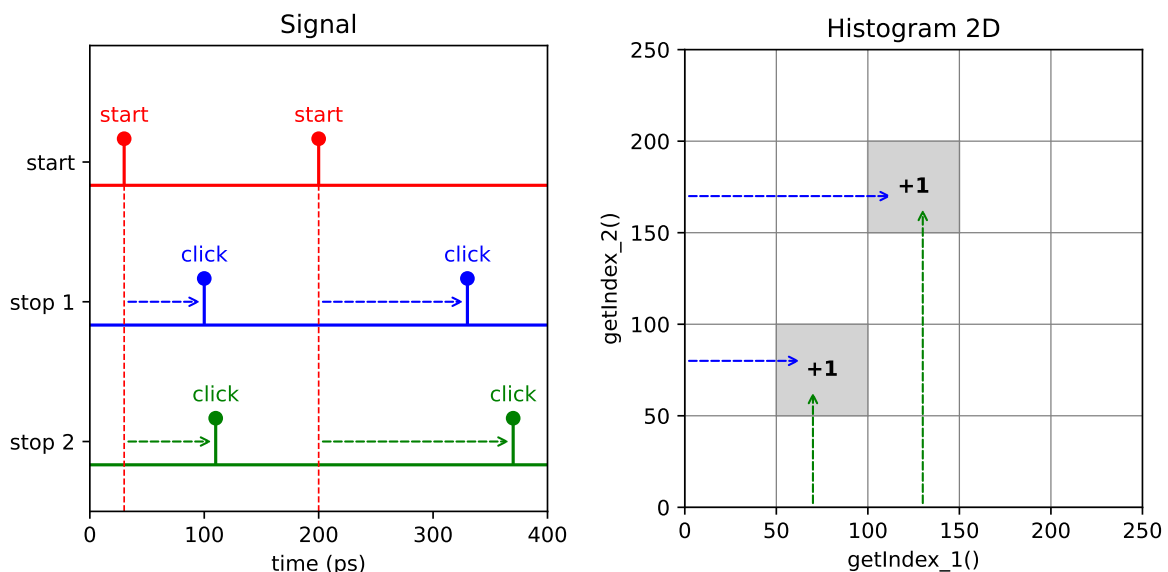
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.38.1 Detailed Description

A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.



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 instance, this kind of measurement is similar to that of typical 2D NMR spectroscopy.

## 9.38.2 Constructor & Destructor Documentation

### 9.38.2.1 Histogram2D()

```
Histogram2D::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,
    int32_t n_bins_1,
    int32_t n_bins_2 )
```

constructor of a [Histogram2D](#) measurement

#### Parameters

<i>tagger</i>	time tagger object
<i>start_channel</i>	channel on which start clicks are received
<i>stop_channel_1</i>	channel on which stop clicks for the time axis 1 are received
<i>stop_channel_2</i>	channel on which stop clicks for the time axis 2 are received
<i>binwidth_1</i>	bin width in ps for the time axis 1
<i>binwidth_2</i>	bin width in ps for the time axis 2
<i>n_bins_1</i>	the number of bins along the time axis 1
<i>n_bins_2</i>	the number of bins along the time axis 2

### 9.38.2.2 ~Histogram2D()

```
Histogram2D::~~Histogram2D ( )
```

## 9.38.3 Member Function Documentation

### 9.38.3.1 clear\_impl()

```
void Histogram2D::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).



### 9.38.3.2 getData()

```
void Histogram2D::getData (
    std::function< int32_t *(size_t, size_t)> array_out )
```

Returns a two-dimensional array of size `n_bins_1` by `n_bins_2` containing the 2D histogram.

### 9.38.3.3 getIndex()

```
void Histogram2D::getIndex (
    std::function< timestamp_t *(size_t, size_t, size_t)> array_out )
```

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

### 9.38.3.4 getIndex\_1()

```
void Histogram2D::getIndex_1 (
    std::function< timestamp_t *(size_t)> array_out )
```

Returns a vector of size `n_bins_1` containing the bin locations in ps for the time axis 1.

### 9.38.3.5 getIndex\_2()

```
void Histogram2D::getIndex_2 (
    std::function< timestamp_t *(size_t)> array_out )
```

Returns a vector of size `n_bins_2` containing the bin locations in ps for the time axis 2.

### 9.38.3.6 next\_impl()

```
bool Histogram2D::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

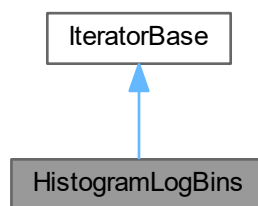
- [Iterators.h](#)

## 9.39 HistogramLogBins Class Reference

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

```
#include <Iterators.h>
```

Inheritance diagram for HistogramLogBins:

**Public Member Functions**

- [HistogramLogBins](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) click\_channel, [channel\\_t](#) start\_channel, double exp\_start, double exp\_stop, int32\_t n\_bins, const [ChannelGate](#) \*click\_gate=nullptr, const [ChannelGate](#) \*start\_gate=nullptr)  
*constructor of a [HistogramLogBins](#) measurement*
- [~HistogramLogBins](#) ()
- [HistogramLogBinsData](#) getDataObject ()
- void [getData](#) (std::function< uint64\_t \*(size\_t)> array\_out)  
*returns the absolute counts for the bins*
- void [getDataNormalizedCountsPerPs](#) (std::function< double \*(size\_t)> array\_out)  
*returns the counts normalized by the binwidth of each bin*
- void [getDataNormalizedG2](#) (std::function< double \*(size\_t)> array\_out)  
*returns the counts normalized by the binwidth and the average count rate.*
- void [getBinEdges](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)  
*returns the edges of the bins in ps*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

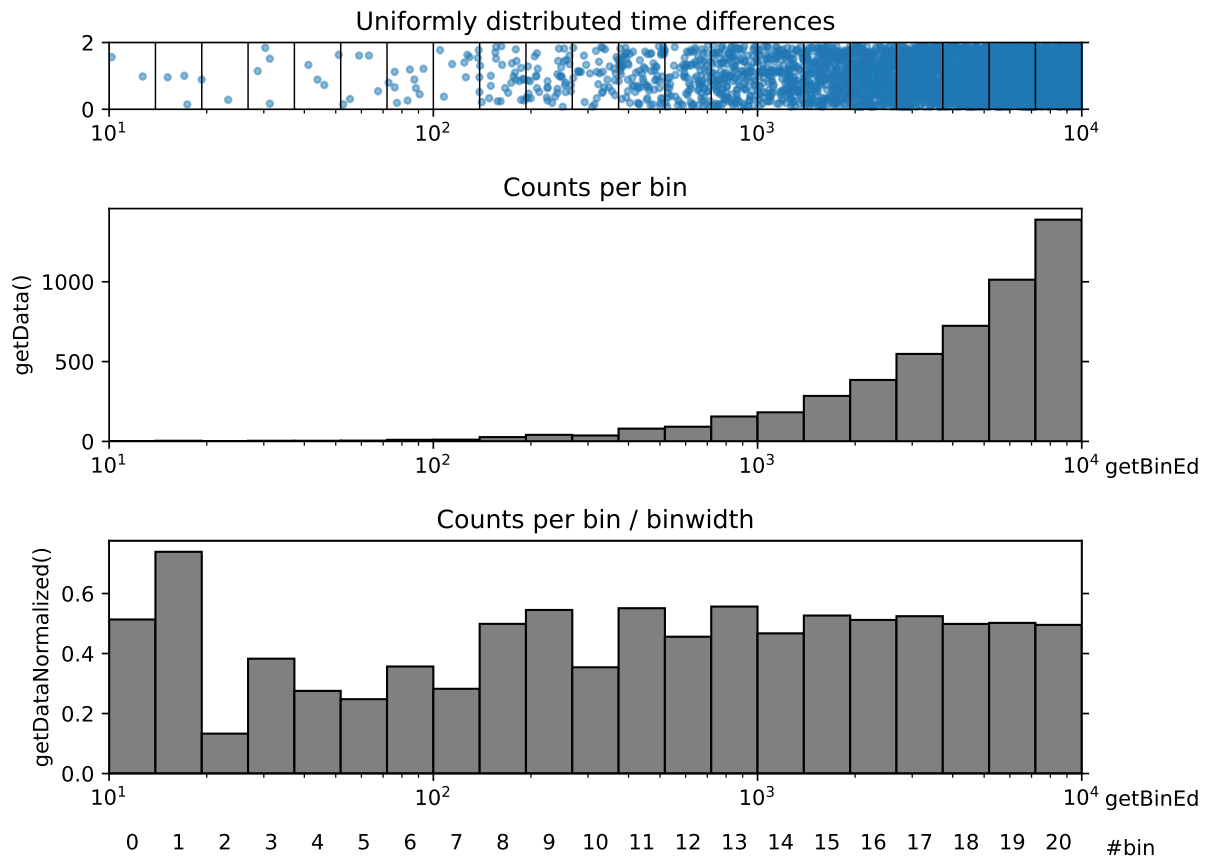
### Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.39.1 Detailed Description

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



This is a multiple start, multiple stop measurement, and works the very same way as the histogram measurement but with logarithmic increasing bin widths. After initializing the measurement (or after an overflow) no data is accumulated in the histogram until the full histogram duration has passed to ensure a balanced count accumulation over the full histogram.

## 9.39.2 Constructor & Destructor Documentation

### 9.39.2.1 HistogramLogBins()

```
HistogramLogBins::HistogramLogBins (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel,
    double exp_start,
    double exp_stop,
    int32_t n_bins,
    const ChannelGate * click_gate = nullptr,
    const ChannelGate * start_gate = nullptr )
```

constructor of a [HistogramLogBins](#) measurement

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel that increments the count in a bin

## Parameters

<i>start_channel</i>	channel that sets start times relative to which clicks on the click channel are measured
<i>exp_start</i>	exponent for the lowest time differences in the histogram: $10^{\text{exp\_start}}$ s, lowest exp_start: -12 => 1ps
<i>exp_stop</i>	exponent for the highest time differences in the histogram: $10^{\text{exp\_stop}}$ s
<i>n_bins</i>	total number of bins in the histogram
<i>click_gate</i>	<a href="#">ChannelGate</a> object for toggling the click_channel, nullptr if unused
<i>start_gate</i>	<a href="#">ChannelGate</a> object for toggling the start_channel, nullptr if unused

**9.39.2.2 ~HistogramLogBins()**

```
HistogramLogBins::~~HistogramLogBins ( )
```

**9.39.3 Member Function Documentation****9.39.3.1 clear\_impl()**

```
void HistogramLogBins::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.39.3.2 getBinEdges()**

```
void HistogramLogBins::getBinEdges (
    std::function< timestamp_t *(size_t)> array_out )
```

returns the edges of the bins in ps

**9.39.3.3 getData()**

```
void HistogramLogBins::getData (
    std::function< uint64_t *(size_t)> array_out )
```

returns the absolute counts for the bins

**9.39.3.4 getDataNormalizedCountsPerPs()**

```
void HistogramLogBins::getDataNormalizedCountsPerPs (
    std::function< double *(size_t)> array_out )
```

returns the counts normalized by the binwidth of each bin

### 9.39.3.5 getDataNormalizedG2()

```
void HistogramLogBins::getDataNormalizedG2 (
    std::function< double *(size_t)> array_out )
```

returns the counts normalized by the binwidth and the average count rate.

This matches the implementation of [Correlation::getDataNormalized](#)

### 9.39.3.6 getDataObject()

```
HistogramLogBinsData HistogramLogBins::getDataObject ( )
```

### 9.39.3.7 next\_impl()

```
bool HistogramLogBins::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.40 HistogramLogBinsData Class Reference

Helper object as return value for [HistogramLogBins::getDataObject](#).

```
#include <Iterators.h>
```

## Public Member Functions

- [~HistogramLogBinsData](#) ( )
- void [getCounts](#) (std::function< uint64\_t \*(size\_t)> array\_out)  
*Get the amount of clicks per bin and per channel.*
- void [getG2Normalization](#) (std::function< double \*(size\_t)> array\_out)  
*Get the calculated normalization for each bin.*
- void [getG2](#) (std::function< double \*(size\_t)> array\_out)  
*Get the normalized histogram.*

## Public Attributes

- const [timestamp\\_t](#) [accumulation\\_time\\_start](#)
- const [timestamp\\_t](#) [accumulation\\_time\\_click](#)

### 9.40.1 Detailed Description

Helper object as return value for [HistogramLogBins::getDataObject](#).

This object stores counts and normalization of the logarithmic bins.

### 9.40.2 Constructor & Destructor Documentation

#### 9.40.2.1 ~HistogramLogBinsData()

```
HistogramLogBinsData::~HistogramLogBinsData ( )
```

### 9.40.3 Member Function Documentation

#### 9.40.3.1 getCounts()

```
void HistogramLogBinsData::getCounts (
    std::function< uint64_t *(size_t)> array_out )
```

Get the amount of clicks per bin and per channel.

#### 9.40.3.2 getG2()

```
void HistogramLogBinsData::getG2 (
    std::function< double *(size_t)> array_out )
```

Get the normalized histogram.



### 9.40.3.3 getG2Normalization()

```
void HistogramLogBinsData::getG2Normalization (
    std::function< double *(size_t)> array_out )
```

Get the calculated normalization for each bin.

## 9.40.4 Member Data Documentation

### 9.40.4.1 accumulation\_time\_click

```
const timestamp_t HistogramLogBinsData::accumulation_time_click
```

### 9.40.4.2 accumulation\_time\_start

```
const timestamp_t HistogramLogBinsData::accumulation_time_start
```

The documentation for this class was generated from the following file:

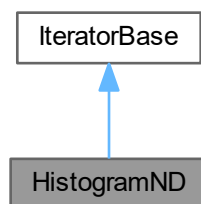
- [Iterators.h](#)

## 9.41 HistogramND Class Reference

A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.

```
#include <Iterators.h>
```

Inheritance diagram for HistogramND:



### Public Member Functions

- [HistogramND](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) start\_channel, std::vector< [channel\\_t](#) > stop\_channels, std::vector< [timestamp\\_t](#) > binwidths, std::vector< int32\_t > n\_bins)  
*constructor of a [Histogram2D](#) measurement*
- [~HistogramND](#) ()
- void [getData](#) (std::function< int32\_t \*(size\_t)> array\_out)
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out, int32\_t dim=0)

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- [std::unique\\_lock](#)< [std::mutex](#) > [getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- [template](#)<typename T >
- void [checkForAbort](#) (T callback)

### Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- [std::set](#)< [channel\\_t](#) > [channels\\_registered](#)
- list of channels used by the iterator*
- bool [running](#)
- running state of the iterator*
- bool [autostart](#)
- Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)
- Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)
- Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)
- For internal use.*
- [std::atomic](#)< bool > [aborting](#)

## 9.41.1 Detailed Description

A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.

This measurement is a N-dimensional version of the [Histogram](#) measurement. The measurement accumulates N-dimensional histogram where stop signals from N separate channels define the bin coordinate. For instance, this kind of measurement is similar to that of typical 2D NMR spectroscopy.

## 9.41.2 Constructor & Destructor Documentation

### 9.41.2.1 HistogramND()

```
HistogramND::HistogramND (
    TimeTaggerBase * tagger,
    channel\_t start\_channel,
    std::vector< channel\_t > stop\_channels,
    std::vector< timestamp\_t > binwidths,
    std::vector< int32\_t > n\_bins )
```

constructor of a [Histogram2D](#) measurement

## Parameters

<i>tagger</i>	time tagger object
<i>start_channel</i>	channel on which start clicks are received
<i>stop_channels</i>	channels on which stop clicks for each time axis are received
<i>binwidths</i>	bin widths in ps for each time axis
<i>n_bins</i>	the number of bins along each time axis

**9.41.2.2 ~HistogramND()**

```
HistogramND::~~HistogramND ( )
```

**9.41.3 Member Function Documentation****9.41.3.1 clear\_impl()**

```
void HistogramND::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.41.3.2 getData()**

```
void HistogramND::getData (
    std::function< int32_t *(size_t)> array_out )
```

Returns a one-dimensional array of size of the product of `n_bins` containing the N-dimensional histogram. The 1D return value is in row-major ordering like on C, Python, C#. This conflicts with Fortran or Matlab. Please reshape the result to get the N-dimensional array.

**9.41.3.3 getIndex()**

```
void HistogramND::getIndex (
    std::function< timestamp_t *(size_t)> array_out,
    int32_t dim = 0 )
```

Returns a vector of size `n_bins[dim]` containing the bin locations in ps for the corresponding time axis.

**9.41.3.4 next\_impl()**

```
bool HistogramND::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.42 HistogramNDImpl< T > Class Template Reference

The documentation for this class was generated from the following file:

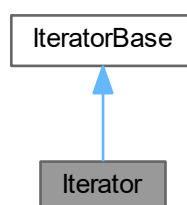
- [Iterators.h](#)

## 9.43 Iterator Class Reference

a deprecated simple event queue

```
#include <Iterators.h>
```

Inheritance diagram for Iterator:



## Public Member Functions

- [Iterator](#) ([TimeTaggerBase](#) \*[tagger](#), [channel\\_t](#) channel)  
*standard constructor*
- [~Iterator](#) ()
- [timestamp\\_t](#) [next](#) ()  
*get next timestamp*
- [uint64\\_t](#) [size](#) ()  
*get queue size*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- [std::string](#) [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) ([std::vector](#)< [Tag](#) > &[incoming\\_tags](#), [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#)) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.43.1 Detailed Description

a deprecated simple event queue

A simple [iterator](#), just keeping a first-in first-out queue of event timestamps.

### 9.43.2 Constructor & Destructor Documentation

#### 9.43.2.1 Iterator()

```
Iterator::Iterator (
    TimeTaggerBase * tagger,
    channel_t channel )
```

standard constructor

Parameters

<i>tagger</i>	the backend
<i>channel</i>	the channel to get events from

#### 9.43.2.2 ~Iterator()

```
Iterator::~~Iterator ( )
```

### 9.43.3 Member Function Documentation

#### 9.43.3.1 clear\_impl()

```
void Iterator::clear_impl ( ) [override], [protected], [virtual]
```

clear [iterator](#) state.

Each [iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

#### 9.43.3.2 next()

```
timestamp_t Iterator::next ( )
```

get next timestamp

get the next timestamp from the queue.



### 9.43.3.3 next\_impl()

```
bool Iterator::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.43.3.4 size()

```
uint64_t Iterator::size ( )
```

get queue size

The documentation for this class was generated from the following file:

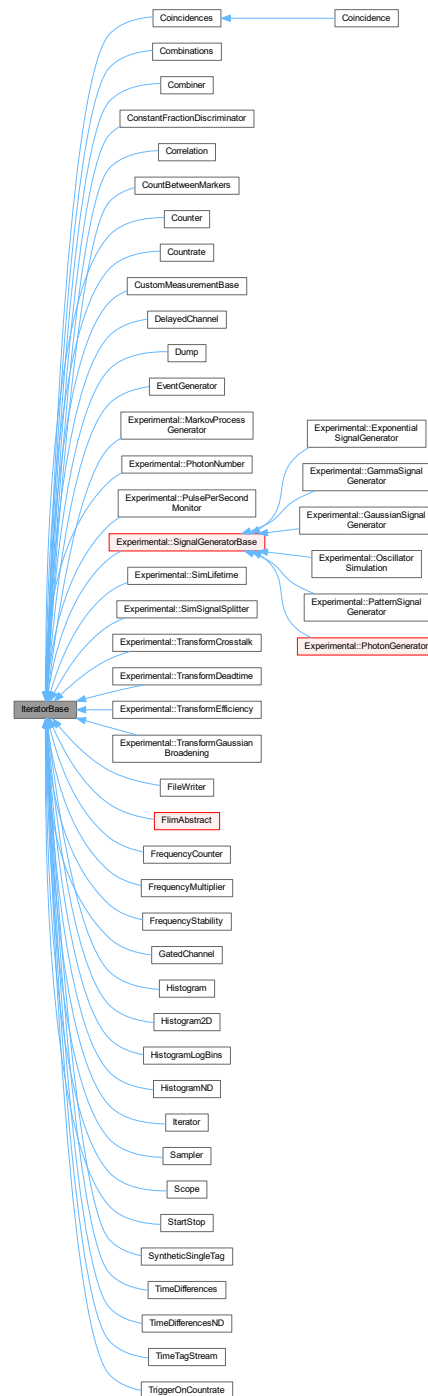
- [Iterators.h](#)

## 9.44 IteratorBase Class Reference

Base class for all iterators.

```
#include <TimeTagger.h>
```

Inheritance diagram for IteratorBase:



## Classes

- class [AbortError](#)

*A custom runtime error thrown by the `abort` call. This can be caught and handled by measurement classes, including `CustomMeasurement`, to perform actions within the abortion process.*

## Public Member Functions

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- [IteratorBase](#) ([TimeTaggerBase](#) \*[tagger](#), std::string [base\\_type](#)="IteratorBase", std::string [extra\\_info](#)="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) [channel](#))  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) [channel](#))  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &[pipeline](#))  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*

- virtual bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time)=0  
*update iterator state*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Protected Attributes

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

## 9.44.1 Detailed Description

Base class for all iterators.

## 9.44.2 Constructor & Destructor Documentation

### 9.44.2.1 IteratorBase()

```
IteratorBase::IteratorBase (
    TimeTaggerBase * tagger,
    std::string base_type_ = "IteratorBase",
    std::string extra_info_ = "" ) [protected]
```

Standard constructor, which will register with the Time Tagger backend.

### 9.44.2.2 ~IteratorBase()

```
virtual IteratorBase::~~IteratorBase ( ) [virtual]
```

destructor, will unregister from the Time Tagger prior finalization.

### 9.44.3 Member Function Documentation

#### 9.44.3.1 abort()

```
void IteratorBase::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.

#### 9.44.3.2 checkForAbort() [1/2]

```
void IteratorBase::checkForAbort ( ) [inline], [protected]
```

#### 9.44.3.3 checkForAbort() [2/2]

```
template<typename T >
void IteratorBase::checkForAbort (
    T callback ) [inline], [protected]
```

#### 9.44.3.4 clear()

```
void IteratorBase::clear ( )
```

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

#### 9.44.3.5 clear\_impl()

```
virtual void IteratorBase::clear_impl ( ) [inline], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the `clear_impl()` method to reset its internal state. The `clear_impl()` function is guarded by the update lock.

Reimplemented in [Combiner](#), [CountBetweenMarkers](#), [Counter](#), [Countrate](#), [TriggerOnCountrate](#), [GatedChannel](#), [FrequencyMultiplier](#), [Iterator](#), [TimeTagStream](#), [Dump](#), [StartStop](#), [TimeDifferences](#), [Histogram2D](#), [HistogramND](#), [TimeDifferencesND](#), [Histogram](#), [FrequencyCounter](#), [HistogramLogBins](#), [Correlation](#), [Scope](#), [FileWriter](#), [EventGenerator](#), [Combinations](#), [CustomMeasurementBase](#), [FlimAbstract](#), [Flim](#), [Sampler](#), [FrequencyStability](#), [Experimental::PulsePerSecondMonitor](#), and [Experimental::PhotonNumber](#).

#### 9.44.3.6 finish\_running()

```
void IteratorBase::finish_running ( ) [protected]
```

Callback for the measurement to stop itself.

It shall only be called while the measurement mutex is locked. It will make sure that no new data is passed to this measurement. The caller has to call `on_stop` themselves if needed.

#### 9.44.3.7 finishInitialization()

```
void IteratorBase::finishInitialization ( ) [protected]
```

method to call after finishing the initialization of the measurement

#### 9.44.3.8 getCaptureDuration()

```
timestamp_t IteratorBase::getCaptureDuration ( )
```

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

##### Returns

Capture duration in ps

#### 9.44.3.9 getConfiguration()

```
std::string IteratorBase::getConfiguration ( )
```

Fetches the overall configuration status of the measurement.

##### Returns

a JSON serialized string with all configuration and status flags.

#### 9.44.3.10 getLock()

```
std::unique_lock< std::mutex > IteratorBase::getLock ( ) [protected]
```

acquire update lock

All mutable operations on a iterator are guarded with an update mutex. Implementers are advised to lock an iterator, whenever internal state is queried or changed.

##### Returns

a lock object, which releases the lock when this instance is freed

#### 9.44.3.11 `getNewVirtualChannel()`

```
channel_t IteratorBase::getNewVirtualChannel ( ) [protected]
```

allocate a new virtual output channel for this iterator

#### 9.44.3.12 `isRunning()`

```
bool IteratorBase::isRunning ( )
```

Returns True if the measurement is collecting the data.

This method will returns False if the measurement was stopped manually by calling `stop()` or automatically after calling `startFor()` and the duration has passed.

##### Note

All measurements start accumulating data immediately after their creation.

##### Returns

True if the measurement is still running

#### 9.44.3.13 `lock()`

```
void IteratorBase::lock ( ) [protected]
```

acquire update lock

All mutable operations on a iterator are guarded with an update mutex. Implementers are advised to `lock()` an iterator, whenever internal state is queried or changed.

#### 9.44.3.14 `next_impl()`

```
virtual bool IteratorBase::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [protected], [pure virtual]
```

update iterator state

Each `Iterator` must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each `Tag` on each registered channel to this callback function.

##### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implemented in [Combiner](#), [CountBetweenMarkers](#), [Counter](#), [Coincidences](#), [Countrate](#), [DelayedChannel](#), [TriggerOnCountrate](#), [GatedChannel](#), [FrequencyMultiplier](#), [Iterator](#), [TimeTagStream](#), [Dump](#), [StartStop](#), [TimeDifferences](#), [Histogram2D](#), [HistogramND](#), [TimeDifferencesND](#), [Histogram](#), [FrequencyCounter](#), [HistogramLogBins](#), [Correlation](#), [Scope](#), [ConstantFractionDiscriminator](#), [FileWriter](#), [EventGenerator](#), [Combinations](#), [CustomMeasurementBase](#), [FlimAbstract](#), [Sampler](#), [SyntheticSingleTag](#), [FrequencyStability](#), [Experimental::PulsePerSecondMonitor](#), [Experimental::SignalGeneratorBase](#), [Experimental::MarkovProcessGenerator](#), [Experimental::SimSignalSplitter](#), [Experimental::TransformEfficiency](#), [Experimental::TransformGaussianBroadening](#), [Experimental::TransformDeadtime](#), [Experimental::TransformCrosstalk](#), [Experimental::SimLifetime](#), and [Experimental::PhotonNumber](#).

**9.44.3.15 on\_start()**

```
virtual void IteratorBase::on_start ( ) [inline], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented in [Counter](#), [Countrate](#), [DelayedChannel](#), [TriggerOnCountrate](#), [Dump](#), [StartStop](#), [TimeDifferences](#), [TimeDifferencesND](#), [Histogram](#), [FrequencyCounter](#), [ConstantFractionDiscriminator](#), [FileWriter](#), [EventGenerator](#), [CustomMeasurementBase](#), [FlimAbstract](#), [Sampler](#), [FrequencyStability](#), and [Experimental::PulsePerSecondMonitor](#).

**9.44.3.16 on\_stop()**

```
virtual void IteratorBase::on_stop ( ) [inline], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented in [Dump](#), [FileWriter](#), [CustomMeasurementBase](#), [Experimental::SignalGeneratorBase](#), and [Experimental::MarkovProcessGenerator](#).

**9.44.3.17 parallelize()**

```
OrderedBarrier::OrderInstance IteratorBase::parallelize (
    OrderedPipeline & pipeline ) [protected]
```

release lock and continue work in parallel

The measurement's lock is released, allowing this measurement to continue, while still executing work in parallel.

**Returns**

a ordered barrier instance that can be synced afterwards.

**9.44.3.18 registerChannel()**

```
void IteratorBase::registerChannel (
    channel_t channel ) [protected]
```

register a channel

Only channels registered by any iterator attached to a backend are delivered over the usb.



## Parameters

<i>channel</i>	the channel
----------------	-------------

**9.44.3.19 start()**

```
void IteratorBase::start ( )
```

Starts or continues data acquisition.

This method is implicitly called when a measurement object is created.

**9.44.3.20 startFor()**

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

Starts or continues the data acquisition for the given duration.

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

<i>capture_duration</i>	capture duration in picoseconds until the measurement is stopped
<i>clear</i>	resets the data acquired

**9.44.3.21 stop()**

```
void IteratorBase::stop ( )
```

After calling this method, the measurement will stop processing incoming tags.

Use [start\(\)](#) or [startFor\(\)](#) to continue or restart the measurement.

**9.44.3.22 unlock()**

```
void IteratorBase::unlock ( ) [protected]
```

release update lock

see [lock\(\)](#)

**9.44.3.23 unregisterChannel()**

```
void IteratorBase::unregisterChannel (
    channel_t channel ) [protected]
```

unregister a channel

## Parameters

<i>channel</i>	the channel
----------------	-------------

**9.44.3.24 waitUntilFinished()**

```
bool IteratorBase::waitUntilFinished (
    int64_t timeout = -1 )
```

Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).

waitUntilFinished will wait according to the timeout and return true if the iterator finished or false if not. Furthermore, when waitUntilFinished is called on a iterator running indefinitely, it will log an error and return immediately.

## Parameters

<i>timeout</i>	time in milliseconds to wait for the measurements. If negative, wait until finished.
----------------	--

## Returns

True if the measurement has finished, false on timeout

**9.44.4 Member Data Documentation****9.44.4.1 aborting**

```
std::atomic<bool> IteratorBase::aborting [protected]
```

**9.44.4.2 autostart**

```
bool IteratorBase::autostart [protected]
```

Condition if this measurement shall be started by the finishInitialization callback.

**9.44.4.3 capture\_duration**

```
timestamp_t IteratorBase::capture_duration [protected]
```

Duration the iterator has already processed data.

**9.44.4.4 channels\_registered**

```
std::set<channel_t> IteratorBase::channels_registered [protected]
```

list of channels used by the iterator

#### 9.44.4.5 pre\_capture\_duration

`timestamp_t` `IteratorBase::pre_capture_duration` [protected]

For internal use.

#### 9.44.4.6 running

`bool` `IteratorBase::running` [protected]

running state of the iterator

#### 9.44.4.7 tagger

`TimeTaggerBase*` `IteratorBase::tagger` [protected]

Pointer to the corresponding Time Tagger object.

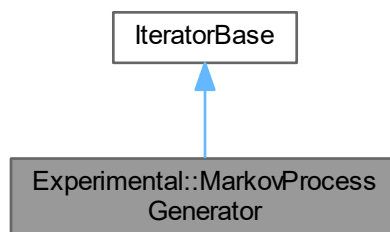
The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.45 Experimental::MarkovProcessGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for `Experimental::MarkovProcessGenerator`:



### Public Member Functions

- [MarkovProcessGenerator](#) ([TimeTaggerBase](#) \*[tagger](#), `uint64_t` `num_states`, `std::vector< double >` `frequencies`, `std::vector< channel_t >` `ref_channels`, `std::vector< channel_t >` `base_channels=std::vector< channel_t >()`, `int32_t` `seed=-1`)  
Construct a continuous-time Markov chain process.
- [~MarkovProcessGenerator](#) ()
- `channel_t` [getChannel](#) ()
- `std::vector< channel_t >` [getChannels](#) ()

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- void [lock](#) ()

- acquire update lock*
- void `unlock` ()
- release update lock*
- `OrderedBarrier::OrderInstance parallelize` (`OrderedPipeline` &pipeline)
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock` ()
- acquire update lock*
- void `finish_running` ()
- Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.45.1 Constructor & Destructor Documentation

### 9.45.1.1 MarkovProcessGenerator()

```
Experimental::MarkovProcessGenerator::MarkovProcessGenerator (
    TimeTaggerBase * tagger,
    uint64_t num_states,
    std::vector< double > frequencies,
    std::vector< channel_t > ref_channels,
    std::vector< channel_t > base_channels = std::vector< channel_t > (),
    int32_t seed = -1 )
```

Construct a continuous-time Markov chain process.

[https://en.wikipedia.org/wiki/Continuous-time\\_Markov\\_chain](https://en.wikipedia.org/wiki/Continuous-time_Markov_chain)

#### Parameters

<i>tagger</i>	reference to a <code>TimeTagger</code>
<i>num_states</i>	Number of exponential states.
<i>frequencies</i>	frequencies of each state transition, it's size is num_states * num_states.
<i>ref_channels</i>	tells the net channel to look at on a state transition. its size is num_states * num_states.
<i>base_channels</i>	channels in which to generate or add the new timetags if CHANNEL_UNUSED or empty, generate a new virtual channel

### 9.45.1.2 ~MarkovProcessGenerator()

```
Experimental::MarkovProcessGenerator::~~MarkovProcessGenerator ( )
```

## 9.45.2 Member Function Documentation

### 9.45.2.1 getChannel()

```
channel_t Experimental::MarkovProcessGenerator::getChannel ( )
```

### 9.45.2.2 getChannels()

```
std::vector< channel_t > Experimental::MarkovProcessGenerator::getChannels ( )
```

### 9.45.2.3 next\_impl()

```
bool Experimental::MarkovProcessGenerator::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.45.2.4 on\_stop()

```
void Experimental::MarkovProcessGenerator::on_stop ( ) [override], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.46 OrderedBarrier Class Reference

Helper for implementing parallel measurements.

```
#include <TimeTagger.h>
```

### Classes

- class [OrderInstance](#)  
*Internal object for serialization.*

### Public Member Functions

- [OrderedBarrier](#) ()
- [~OrderedBarrier](#) ()
- [OrderInstance queue](#) ()
- void [waitUntilFinished](#) ()

### 9.46.1 Detailed Description

Helper for implementing parallel measurements.

### 9.46.2 Constructor & Destructor Documentation

#### 9.46.2.1 OrderedBarrier()

```
OrderedBarrier::OrderedBarrier ( )
```

#### 9.46.2.2 ~OrderedBarrier()

```
OrderedBarrier::~~OrderedBarrier ( )
```

### 9.46.3 Member Function Documentation

#### 9.46.3.1 queue()

```
OrderInstance OrderedBarrier::queue ( )
```

#### 9.46.3.2 waitUntilFinished()

```
void OrderedBarrier::waitUntilFinished ( )
```

The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.47 OrderedPipeline Class Reference

Helper for implementing parallel measurements.

```
#include <TimeTagger.h>
```

### Public Member Functions

- [OrderedPipeline](#) ()
- [~OrderedPipeline](#) ()

### 9.47.1 Detailed Description

Helper for implementing parallel measurements.

### 9.47.2 Constructor & Destructor Documentation

#### 9.47.2.1 OrderedPipeline()

```
OrderedPipeline::OrderedPipeline ( )
```

#### 9.47.2.2 ~OrderedPipeline()

```
OrderedPipeline::~~OrderedPipeline ( )
```

The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.48 OrderedBarrier::OrderInstance Class Reference

Internal object for serialization.

```
#include <TimeTagger.h>
```

### Public Member Functions

- [OrderInstance](#) ()
- [OrderInstance](#) ([OrderedBarrier](#) \*parent, uint64\_t instance\_id)
- [~OrderInstance](#) ()
- void [sync](#) ()
- void [release](#) ()



### 9.48.1 Detailed Description

Internal object for serialization.

### 9.48.2 Constructor & Destructor Documentation

#### 9.48.2.1 OrderInstance() [1/2]

```
OrderedBarrier::OrderInstance::OrderInstance ( )
```

#### 9.48.2.2 OrderInstance() [2/2]

```
OrderedBarrier::OrderInstance::OrderInstance (
    OrderedBarrier * parent,
    uint64_t instance_id )
```

#### 9.48.2.3 ~OrderInstance()

```
OrderedBarrier::OrderInstance::~~OrderInstance ( )
```

### 9.48.3 Member Function Documentation

#### 9.48.3.1 release()

```
void OrderedBarrier::OrderInstance::release ( )
```

#### 9.48.3.2 sync()

```
void OrderedBarrier::OrderInstance::sync ( )
```

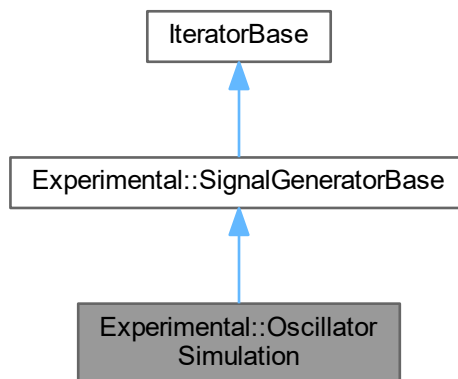
The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.49 Experimental::OscillatorSimulation Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::OscillatorSimulation:



### Public Member Functions

- [OscillatorSimulation](#) ([TimeTaggerBase](#) \*tagger, double nominal\_frequency, double coeff\_phase\_white=0.0, double coeff\_phase\_flicker=0.0, double coeff\_freq\_white=0.0, double coeff\_freq\_flicker=0.0, double coeff\_↵\_random\_drift=0.0, double coeff\_linear\_drift=0.0, [channel\\_t](#) base\_channel=CHANNEL\_UNUSED, int32\_↵\_t seed=-1)  
Construct a simulated oscillator event channel.
- [~OscillatorSimulation](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=CHANNEL\_UNUSED)
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
the new virtual channel

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
destructor, will unregister from the Time Tagger prior finalization.
- void [start](#) ()  
Starts or continues data acquisition.
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
Starts or continues the data acquisition for the given duration.
- bool [waitUntilFinished](#) (int64\_t timeout=-1)

- Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- void [initialize](#) ([timestamp\\_t](#) initial\_time) override
- [timestamp\\_t](#) [get\\_next](#) () override
- void [on\\_restart](#) ([timestamp\\_t](#) restart\_time) override

### Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) begin\_time, [timestamp\\_t](#) end\_time) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool is\_finished)

### Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- void [lock](#) ()  
*acquire update lock*

- void `unlock ()`  
*release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`  
*acquire update lock*
- void `finish_running ()`  
*Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- `template<typename T >`  
void `checkForAbort (T callback)`

### Additional Inherited Members

### Protected Attributes inherited from `Experimental::SignalGeneratorBase`

- `std::unique_ptr< SignalGeneratorBaseImpl > impl`

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t > channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.49.1 Constructor & Destructor Documentation

### 9.49.1.1 OscillatorSimulation()

```
Experimental::OscillatorSimulation::OscillatorSimulation (
    TimeTaggerBase * tagger,
    double nominal_frequency,
    double coeff_phase_white = 0.0,
    double coeff_phase_flicker = 0.0,
    double coeff_freq_white = 0.0,
    double coeff_freq_flicker = 0.0,
    double coeff_random_drift = 0.0,
    double coeff_linear_drift = 0.0,
    channel_t base_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

Construct a simulated oscillator event channel.

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>nominal_frequency</i>	Normal frequency of the oscillator in Hz
<i>coeff_phase_white</i>	RMS value of the white phase noise in seconds.
<i>coeff_phase_flicker</i>	RMS value of the flicker phase noise in seconds per octave.
<i>coeff_freq_white</i>	Scaling parameter for the white frequency modulated noise in sqrt(s), use $10e-12 * \sqrt{1e-3}$ for 10 ppt RMS error at 1 kHz cutoff frequency.
<i>coeff_freq_flicker</i>	Scaling parameter for the relative flicker frequency modulated noise, use $10e-12$ for 10 ppt error per octave.
<i>coeff_random_drift</i>	Scaling parameter for the random walk drift in sqrt(Hz), use $10e-9 / \sqrt{60*60*24}$ for 10 ppb / sqrt(day).
<i>coeff_linear_drift</i>	Scaling parameter for the relative linear frequency drift in Hz, use $1e-6 / (60*60*24*365)$ for 1 ppm / year.
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

## 9.49.1.2 ~OscillatorSimulation()

```
Experimental::OscillatorSimulation::~~OscillatorSimulation ( )
```

## 9.49.2 Member Function Documentation

## 9.49.2.1 get\_next()

```
timestamp_t Experimental::OscillatorSimulation::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

## 9.49.2.2 initialize()

```
void Experimental::OscillatorSimulation::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

## 9.49.2.3 on\_restart()

```
void Experimental::OscillatorSimulation::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

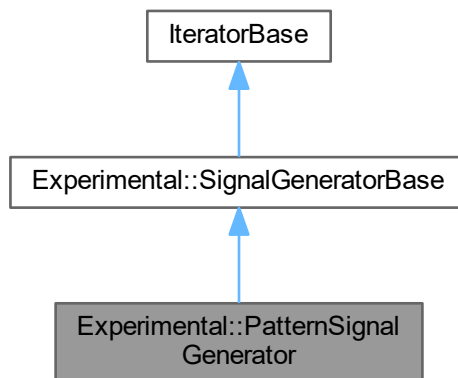
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.50 Experimental::PatternSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::PatternSignalGenerator:



### Public Member Functions

- [PatternSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, std::vector< [timestamp\\_t](#) > sequence, bool repeat=false, [timestamp\\_t](#) start\_delay=0, [timestamp\\_t](#) spacing=0, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))  
Construct a pattern event generator.
- [~PatternSignalGenerator](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
the new virtual channel

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
destructor, will unregister from the Time Tagger prior finalization.
- void [start](#) ()  
Starts or continues data acquisition.
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
Starts or continues the data acquisition for the given duration.
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).
- void [stop](#) ()

After calling this method, the measurement will stop processing incoming tags.

- void [clear](#) ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- void [abort](#) ()  
Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.
- bool [isRunning](#) ()  
Returns True if the measurement is collecting the data.
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
Total capture duration since the measurement creation or last call to [clear\(\)](#).
- std::string [getConfiguration](#) ()  
Fetches the overall configuration status of the measurement.

### Protected Member Functions

- void [initialize](#) ([timestamp\\_t](#) initial\_time) override
- [timestamp\\_t](#) [get\\_next](#) () override
- void [on\\_restart](#) ([timestamp\\_t](#) restart\_time) override

### Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) begin\_time, [timestamp\\_t](#) end\_time) override  
update iterator state
- void [on\\_stop](#) () override  
callback when the measurement class is stopped
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool is\_finished)

### Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
Standard constructor, which will register with the Time Tagger backend.
- void [registerChannel](#) ([channel\\_t](#) channel)  
register a channel
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
unregister a channel
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
allocate a new virtual output channel for this iterator
- void [finishInitialization](#) ()  
method to call after finishing the initialization of the measurement
- virtual void [clear\\_impl](#) ()  
clear *Iterator* state.
- virtual void [on\\_start](#) ()  
callback when the measurement class is started
- void [lock](#) ()  
acquire update lock
- void [unlock](#) ()  
release update lock

- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` [getLock](#) ()  
*acquire update lock*
- `void` [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- `void` [checkForAbort](#) ()
- `template<typename T>`  
`void` [checkForAbort](#) (T callback)

### Additional Inherited Members

### Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- `std::unique_ptr<` [SignalGeneratorBaseImpl](#) `>` [impl](#)

### Protected Attributes inherited from [IteratorBase](#)

- `std::set<` [channel\\_t](#) `>` [channels\\_registered](#)  
*list of channels used by the iterator*
- `bool` [running](#)  
*running state of the iterator*
- `bool` [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- `timestamp_t` [pre\\_capture\\_duration](#)  
*For internal use.*
- `std::atomic< bool >` [aborting](#)

## 9.50.1 Constructor & Destructor Documentation

### 9.50.1.1 [PatternSignalGenerator](#)()

```
Experimental::PatternSignalGenerator::PatternSignalGenerator (
    TimeTaggerBase * tagger,
    std::vector< timestamp_t > sequence,
    bool repeat = false,
    timestamp_t start_delay = 0,
    timestamp_t spacing = 0,
    channel_t base_channel = CHANNEL_UNUSED )
```

Construct a pattern event generator.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>	Generated by Doxygen
<i>sequence</i>	sequence of offsets pattern to be used continuously.	
<i>repeat</i>	tells if to repeat the pattern or only generate it once.	
<i>start_delay</i>	initial delay before the first pattern is applied.	
<i>spacing</i>	delay between pattern repetitions.	
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.	



### 9.50.1.2 ~PatternSignalGenerator()

Experimental::PatternSignalGenerator::~~PatternSignalGenerator ( )

## 9.50.2 Member Function Documentation

### 9.50.2.1 get\_next()

`timestamp_t` Experimental::PatternSignalGenerator::get\_next ( ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

### 9.50.2.2 initialize()

void Experimental::PatternSignalGenerator::initialize (   
 `timestamp_t` *initial\_time* ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

### 9.50.2.3 on\_restart()

void Experimental::PatternSignalGenerator::on\_restart (   
 `timestamp_t` *restart\_time* ) [override], [protected], [virtual]

Reimplemented from [Experimental::SignalGeneratorBase](#).

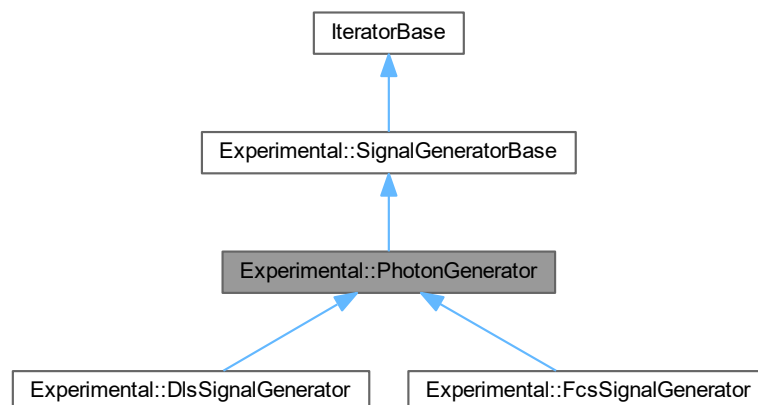
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.51 Experimental::PhotonGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::PhotonGenerator:



## Public Member Functions

- [PhotonGenerator](#) ([TimeTaggerBase](#) \*tagger, double countrate, [channel\\_t](#) base\_channel, int32\_t seed=-1)  
*A generator for TimeTags arising from a laser driven process. [PhotonGenerator](#) should be used as the base class of a virtual class with a dedicated `get_intensity` function which models the relevant physical processes.*
- [~PhotonGenerator](#) ()
- void [finalize\\_init](#) ()
- void [set\\_T\\_PERIOD](#) ([timestamp\\_t](#) new\_T)
- [timestamp\\_t](#) [get\\_T\\_PERIOD](#) ()

## Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- void [initialize](#) ([timestamp\\_t](#) initial\_time) override
- void [on\\_restart](#) ([timestamp\\_t](#) restart\_time) override
- [timestamp\\_t](#) [get\\_next](#) () override
- virtual double [get\\_intensity](#) ()=0

## Protected Member Functions inherited from Experimental::SignalGeneratorBase

- bool `next_impl` (std::vector< Tag > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void `on_stop` () override  
*callback when the measurement class is stopped*
- bool `isProcessingFinished` ()
- void `set_processing_finished` (bool is\_finished)

## Protected Member Functions inherited from IteratorBase

- `IteratorBase` (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (channel\_t channel)  
*register a channel*
- void `unregisterChannel` (channel\_t channel)  
*unregister a channel*
- channel\_t `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear Iterator state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- OrderedBarrier::OrderInstance `parallelize` (OrderedPipeline &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

## Protected Attributes

- timestamp\_t T\_PERIOD

## Protected Attributes inherited from Experimental::SignalGeneratorBase

- std::unique\_ptr< SignalGeneratorBaseImpl > impl

## Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the `finishInitialization` callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.51.1 Constructor & Destructor Documentation

### 9.51.1.1 [PhotonGenerator\(\)](#)

```
Experimental::PhotonGenerator::PhotonGenerator (
    TimeTaggerBase * tagger,
    double countrate,
    channel\_t base_channel,
    int32_t seed = -1 )
```

A generator for TimeTags arising from a laser driven process. [PhotonGenerator](#) should be used as the base class of a virtual class with a dedicated `get_intensity` function which models the relevant physical processes.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a> .
<i>countrate</i>	rate (in Hz) of Time Tags to be generated.
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.51.1.2 [~PhotonGenerator\(\)](#)

```
Experimental::PhotonGenerator::~~PhotonGenerator ( )
```

## 9.51.2 Member Function Documentation

### 9.51.2.1 [finalize\\_init\(\)](#)

```
void Experimental::PhotonGenerator::finalize_init ( )
```

### 9.51.2.2 get\_intensity()

```
virtual double Experimental::PhotonGenerator::get_intensity ( ) [protected], [pure virtual]
```

Implemented in [Experimental::DlsSignalGenerator](#), and [Experimental::FcsSignalGenerator](#).

### 9.51.2.3 get\_next()

```
timestamp_t Experimental::PhotonGenerator::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.51.2.4 get\_T\_PERIOD()

```
timestamp_t Experimental::PhotonGenerator::get_T_PERIOD ( )
```

### 9.51.2.5 initialize()

```
void Experimental::PhotonGenerator::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.51.2.6 on\_restart()

```
void Experimental::PhotonGenerator::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

### 9.51.2.7 set\_T\_PERIOD()

```
void Experimental::PhotonGenerator::set_T_PERIOD (
    timestamp_t new_T )
```

## 9.51.3 Member Data Documentation

### 9.51.3.1 T\_PERIOD

```
timestamp_t Experimental::PhotonGenerator::T_PERIOD [protected]
```

The documentation for this class was generated from the following file:

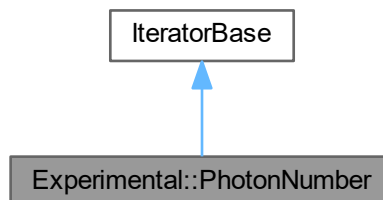
- [Iterators.h](#)

## 9.52 Experimental::PhotonNumber Class Reference

Photon number resolution.

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::PhotonNumber:



### Public Member Functions

- `PhotonNumber` (`TimeTaggerBase` \*tagger, `channel_t` trigger\_ch, `channel_t` signal\_start\_ch, `channel_t` signal\_stop\_ch, double slope, `std::vector< double >` x\_intercepts, `timestamp_t` dead\_time)  
construct a *PhotonNumber*
- `~PhotonNumber` ()
- `std::vector< channel_t >` const & `getChannels` () const  
the new virtual channels

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
destructor, will unregister from the Time Tagger prior finalization.
- void `start` ()  
Starts or continues data acquisition.
- void `startFor` (`timestamp_t` capture\_duration, bool clear=true)  
Starts or continues the data acquisition for the given duration.
- bool `waitUntilFinished` (int64\_t timeout=-1)  
Blocks the execution until the measurement has finished. Can be used with `startFor()`.
- void `stop` ()  
After calling this method, the measurement will stop processing incoming tags.
- void `clear` ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- void `abort` ()  
Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.
- bool `isRunning` ()  
Returns True if the measurement is collecting the data.
- `timestamp_t` `getCaptureDuration` ()  
Total capture duration since the measurement creation or last call to `clear()`.
- `std::string` `getConfiguration` ()  
Fetches the overall configuration status of the measurement.

**Protected Member Functions**

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*

**Protected Member Functions inherited from `IteratorBase`**

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

**Additional Inherited Members****Protected Attributes inherited from `IteratorBase`**

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

### 9.52.1 Detailed Description

Photon number resolution.

### 9.52.2 Constructor & Destructor Documentation

#### 9.52.2.1 PhotonNumber()

```
Experimental::PhotonNumber::PhotonNumber (
    TimeTaggerBase * tagger,
    channel_t trigger_ch,
    channel_t signal_start_ch,
    channel_t signal_stop_ch,
    double slope,
    std::vector< double > x_intercepts,
    timestamp_t dead_time )
```

construct a [PhotonNumber](#)

##### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>trigger_ch</i>	trigger channel
<i>signal_start_ch</i>	start-of-signal channel (likely rising edge)
<i>signal_stop_ch</i>	end-of-signal channel (likely falling edge)
<i>slope</i>	common slope of decision boundary lines
<i>x_intercepts</i>	x-intercepts of decision boundary lines. Has to be in descending order
<i>dead_time</i>	the dead time of the detector

#### 9.52.2.2 ~PhotonNumber()

```
Experimental::PhotonNumber::~~PhotonNumber ( )
```

### 9.52.3 Member Function Documentation

#### 9.52.3.1 clear\_impl()

```
void Experimental::PhotonNumber::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).



### 9.52.3.2 getChannels()

```
std::vector< channel_t > const & Experimental::PhotonNumber::getChannels ( ) const
```

the new virtual channels

This function returns the IDs of the allocated virtual channels, corresponding to photon counts of  $1, \dots, N, \geq N+1$  for given  $N$  decision boundary lines.

### 9.52.3.3 next\_impl()

```
bool Experimental::PhotonNumber::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.53 Experimental::PulsePerSecondData Class Reference

Helper object as return value for [PulsePerSecondMonitor::getDataObject](#).

```
#include <Iterators.h>
```

## Public Member Functions

- void [getIndices](#) (std::function< int64\_t \*(size\_t)> array\_out)  
*The indices of each reference pulse in the [PulsePerSecondData](#) object. In case of overflows in the reference channel, this index will be incremented by the number of missed pulses.*
- void [getReferenceOffsets](#) (std::function< double \*(size\_t)> array\_out)  
*A list of offsets of each reference pulse with respect to its predecessor, with the period subtracted.*
- void [getSignalOffsets](#) (std::function< double \*(size\_t, size\_t)> array\_out)  
*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`.*
- void [getUtcSeconds](#) (std::function< double \*(size\_t)> array\_out)  
*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.*
- std::vector< std::string > [getUtcDates](#) ()  
*The UTC timestamps for the system time at which each reference pulse is processed, as a string with ISO 8601 formatting.*
- void [getStatus](#) (std::function< bool \*(size\_t)> array\_out)  
*A vector of booleans values describing whether all signals, including from the reference source, were detected.*
- [~PulsePerSecondData](#) ()

## Public Attributes

- const size\_t [size](#)  
*Number of reference pulses contained in the [PulsePerSecondData](#) object.*

## 9.53.1 Detailed Description

Helper object as return value for [PulsePerSecondMonitor::getDataObject](#).

This object stores the results of all monitored PPS pulses.

## 9.53.2 Constructor & Destructor Documentation

### 9.53.2.1 [~PulsePerSecondData\(\)](#)

```
Experimental::PulsePerSecondData::~~PulsePerSecondData ( )
```

## 9.53.3 Member Function Documentation

### 9.53.3.1 [getIndices\(\)](#)

```
void Experimental::PulsePerSecondData::getIndices (
    std::function< int64_t *(size_t)> array_out )
```

The indices of each reference pulse in the [PulsePerSecondData](#) object. In case of overflows in the reference channel, this index will be incremented by the number of missed pulses.

### 9.53.3.2 getReferenceOffsets()

```
void Experimental::PulsePerSecondData::getReferenceOffsets (
    std::function< double *(size_t)> array_out )
```

A list of offsets of each reference pulse with respect to its predecessor, with the period subtracted.

### 9.53.3.3 getSignalOffsets()

```
void Experimental::PulsePerSecondData::getSignalOffsets (
    std::function< double *(size_t, size_t)> array_out )
```

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`.

### 9.53.3.4 getStatus()

```
void Experimental::PulsePerSecondData::getStatus (
    std::function< bool *(size_t)> array_out )
```

A vector of boolean values describing whether all signals, including from the reference source, were detected.

### 9.53.3.5 getUtcDates()

```
std::vector< std::string > Experimental::PulsePerSecondData::getUtcDates ( )
```

The UTC timestamps for the system time at which each reference pulse is processed, as a string with ISO 8601 formatting.

### 9.53.3.6 getUtcSeconds()

```
void Experimental::PulsePerSecondData::getUtcSeconds (
    std::function< double *(size_t)> array_out )
```

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.

## 9.53.4 Member Data Documentation

### 9.53.4.1 size

```
const size_t Experimental::PulsePerSecondData::size
```

Number of reference pulses contained in the [PulsePerSecondData](#) object.

The documentation for this class was generated from the following file:

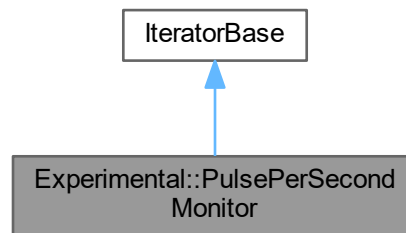
- [Iterators.h](#)

## 9.54 Experimental::PulsePerSecondMonitor Class Reference

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

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::PulsePerSecondMonitor:



### Public Member Functions

- [PulsePerSecondMonitor](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) reference\_channel, [std::vector](#)< [channel\\_t](#) > signal\_channels, [std::string](#) filename="", [timestamp\\_t](#) period=1E12)  
*constructor of a [PulsePerSecondMonitor](#) measurement*
- [~PulsePerSecondMonitor](#) ()
- [PulsePerSecondData](#) [getDataObject](#) (bool remove=false)  
*Fetches the results of all measured PPS pulses.*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- [std::string](#) [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

**Protected Member Functions**

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*
- void `on_start` () override  
*callback when the measurement class is started*

**Protected Member Functions inherited from `IteratorBase`**

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

**Additional Inherited Members****Protected Attributes inherited from `IteratorBase`**

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`

### 9.54.1 Detailed Description

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

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).

### 9.54.2 Constructor & Destructor Documentation

#### 9.54.2.1 PulsePerSecondMonitor()

```
Experimental::PulsePerSecondMonitor::PulsePerSecondMonitor (
    TimeTaggerBase * tagger,
    channel_t reference_channel,
    std::vector< channel_t > signal_channels,
    std::string filename = "",
    timestamp_t period = 1E12 )
```

constructor of a [PulsePerSecondMonitor](#) measurement

##### Parameters

<i>tagger</i>	a <a href="#">TimeTagger</a> object.
<i>reference_channel</i>	the channel whose signal will be the standard against which other signals are compared.
<i>signal_channels</i>	a list of channel numbers with PPS signals to be compared to the reference.
<i>filename</i>	the name of the .csv file to store measurement data. By default, no data is written to file.
<i>period</i>	the assumed period of the reference source, typically one second, in picoseconds.

#### 9.54.2.2 ~PulsePerSecondMonitor()

```
Experimental::PulsePerSecondMonitor::~~PulsePerSecondMonitor ( )
```

### 9.54.3 Member Function Documentation

#### 9.54.3.1 clear\_impl()

```
void Experimental::PulsePerSecondMonitor::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.54.3.2 getDataObject()

```
PulsePerSecondData Experimental::PulsePerSecondMonitor::getDataObject (
    bool remove = false )
```

Fetches the results of all measured PPS pulses.

#### Returns

a [PulsePerSecondData](#) object, which contains all data of the monitored PPS pulses. To remove the data from the internal memory after each call, set `remove` to `true`.

### 9.54.3.3 next\_impl()

```
bool Experimental::PulsePerSecondMonitor::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.54.3.4 on\_start()

```
void Experimental::PulsePerSecondMonitor::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

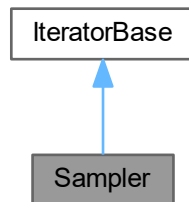
- [Iterators.h](#)

## 9.55 Sampler Class Reference

a triggered sampling measurement

```
#include <Iterators.h>
```

Inheritance diagram for Sampler:



### Public Member Functions

- [Sampler](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) trigger, std::vector< [channel\\_t](#) > channels, size\_t max\_  
triggers)  
*constructor of a [Sampler](#) measurement*
- [~Sampler](#) ()
- void [getData](#) (std::function< [timestamp\\_t](#) \*(size\_t, size\_t)> array\_out)  
*fetches the internal data as 2D array.*
- void [getDataAsMask](#) (std::function< [timestamp\\_t](#) \*(size\_t, size\_t)> array\_out)  
*fetches the internal data as 2D array with a channel mask.*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*



- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (`std::vector`< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*
- void `on_start` () override  
*callback when the measurement class is started*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock`< `std::mutex` > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- `template`< `typename` T >  
void `checkForAbort` (T callback)

## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the `finishInitialization` callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.55.1 Detailed Description

a triggered sampling measurement

This measurement class will perform a triggered sampling measurement. So 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 state without any deadtime. So every event will be recorded exactly once.

The unknown state might happen after an overflow without an event on the input channel. This processing assumes that no event was filtered by the deadtime. Else invalid data will be reported till the next event on this input channel.

## 9.55.2 Constructor & Destructor Documentation

### 9.55.2.1 [Sampler\(\)](#)

```
Sampler::Sampler (
    TimeTaggerBase * tagger,
    channel\_t trigger,
    std::vector< channel\_t > channels,
    size_t max\_triggers )
```

constructor of a [Sampler](#) measurement

#### Parameters

<i><a href="#">tagger</a></i>	reference to a <a href="#">TimeTagger</a>
<i><a href="#">trigger</a></i>	the channel which shall trigger the measurement
<i><a href="#">channels</a></i>	a list of channels which will be recorded for every trigger
<i><a href="#">max_triggers</a></i>	the maximum amount of triggers without <code>getData*</code> call till this measurement will stop itself

### 9.55.2.2 ~Sampler()

```
Sampler::~~Sampler ( )
```

## 9.55.3 Member Function Documentation

### 9.55.3.1 clear\_impl()

```
void Sampler::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.55.3.2 getData()

```
void Sampler::getData (
    std::function< timestamp\_t *(size_t, size_t)> array_out )
```

fetches the internal data as 2D array.

Its layout is roughly: [ [timestamp of first trigger, state of channel 0, state of channel 1, ...], [timestamp of second trigger, state of channel 0, state of channel 1, ...], ... ] Where state means: 0 – low 1 – high 2 – undefined (after overflow)

### 9.55.3.3 getDataAsMask()

```
void Sampler::getDataAsMask (
    std::function< timestamp\_t *(size_t, size_t)> array_out )
```

fetches the internal data as 2D array with a channel mask.

Its layout is roughly: [ [timestamp of first trigger, (state of channel 0) << 0 | (state of channel 1) << 1 | ... | undefined << 63], [timestamp of second trigger, (state of channel 0) << 0 | (state of channel 1) << 1 | ... | undefined << 63], ... ] Where state means: 0 – low or undefined (after overflow) 1 – high

### 9.55.3.4 next\_impl()

```
bool Sampler::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp\_t begin_time,
    timestamp\_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.55.3.5 on\_start()**

```
void Sampler::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

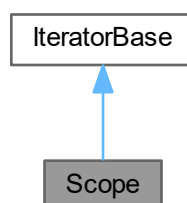
- [Iterators.h](#)

**9.56 Scope Class Reference**

a scope measurement

```
#include <Iterators.h>
```

Inheritance diagram for Scope:



## Public Member Functions

- `Scope (TimeTaggerBase *tagger, std::vector< channel_t > event_channels, channel_t trigger_channel, timestamp_t window_size=1000000000, int32_t n_traces=1, int32_t n_max_events=1000)`  
*constructor of a `Scope` measurement*
- `~Scope ()`
- `bool ready ()`
- `int32_t triggered ()`
- `std::vector< std::vector< Event > > getData ()`
- `timestamp_t getWindowSize ()`

## Public Member Functions inherited from `IteratorBase`

- `virtual ~IteratorBase ()`  
*destructor, will unregister from the Time Tagger prior finalization.*
- `void start ()`  
*Starts or continues data acquisition.*
- `void startFor (timestamp_t capture_duration, bool clear=true)`  
*Starts or continues the data acquisition for the given duration.*
- `bool waitUntilFinished (int64_t timeout=-1)`  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- `void stop ()`  
*After calling this method, the measurement will stop processing incoming tags.*
- `void clear ()`  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- `void abort ()`  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- `bool isRunning ()`  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration ()`  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration ()`  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- `bool next_impl (std::vector< Tag > &incoming_tags, timestamp_t begin_time, timestamp_t end_time)` override  
*update iterator state*
- `void clear_impl ()` override  
*clear `Iterator` state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
Standard constructor, which will register with the Time Tagger backend.
- void [registerChannel](#) ([channel\\_t](#) channel)  
register a channel
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
unregister a channel
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
allocate a new virtual output channel for this iterator
- void [finishInitialization](#) ()  
method to call after finishing the initialization of the measurement
- virtual void [on\\_start](#) ()  
callback when the measurement class is started
- virtual void [on\\_stop](#) ()  
callback when the measurement class is stopped
- void [lock](#) ()  
acquire update lock
- void [unlock](#) ()  
release update lock
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
release lock and continue work in parallel
- std::unique\_lock< std::mutex > [getLock](#) ()  
acquire update lock
- void [finish\\_running](#) ()  
Callback for the measurement to stop itself.
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

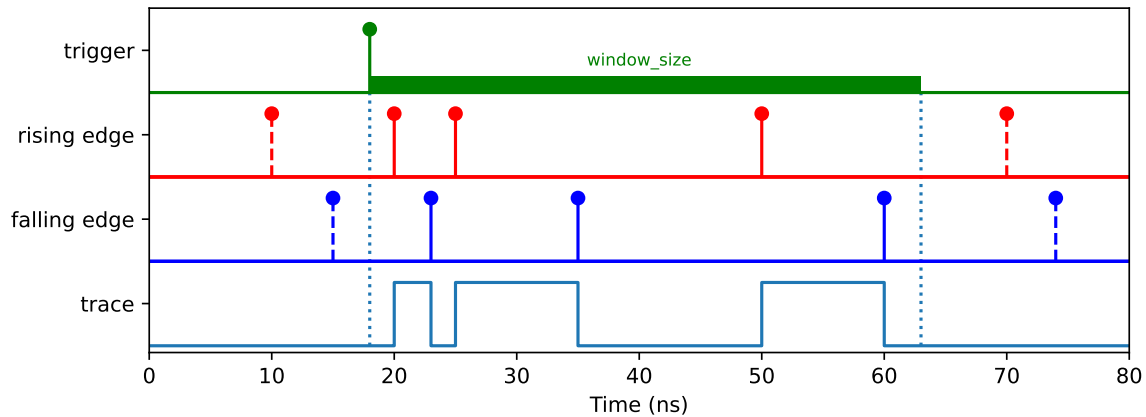
## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
list of channels used by the iterator
- bool [running](#)  
running state of the iterator
- bool [autostart](#)  
Condition if this measurement shall be started by the finishInitialization callback.
- [TimeTaggerBase](#) \* [tagger](#)  
Pointer to the corresponding Time Tagger object.
- [timestamp\\_t](#) [capture\\_duration](#)  
Duration the iterator has already processed data.
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
For internal use.
- std::atomic< bool > [aborting](#)

### 9.56.1 Detailed Description

a scope measurement



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.

### 9.56.2 Constructor & Destructor Documentation

#### 9.56.2.1 Scope()

```
Scope::Scope (
    TimeTaggerBase * tagger,
    std::vector< channel_t > event_channels,
    channel_t trigger_channel,
    timestamp_t window_size = 1000000000,
    int32_t n_traces = 1,
    int32_t n_max_events = 1000 )
```

constructor of a [Scope](#) measurement

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>event_channels</i>	channels which are captured
<i>trigger_channel</i>	channel that starts a new trace
<i>window_size</i>	window time of each trace
<i>n_traces</i>	amount of traces ( <code>n_traces &lt; 1</code> , automatic retrigger)
<i>n_max_events</i>	maximum number of tags in each trace

### 9.56.2.2 ~Scope()

```
Scope::~~Scope ( )
```

## 9.56.3 Member Function Documentation

### 9.56.3.1 clear\_impl()

```
void Scope::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.56.3.2 getData()

```
std::vector< std::vector< Event > > Scope::getData ( )
```

### 9.56.3.3 getWindowSize()

```
timestamp_t Scope::getWindowSize ( )
```

### 9.56.3.4 next\_impl()

```
bool Scope::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise



Implements [IteratorBase](#).

### 9.56.3.5 ready()

```
bool Scope::ready ( )
```

### 9.56.3.6 triggered()

```
int32_t Scope::triggered ( )
```

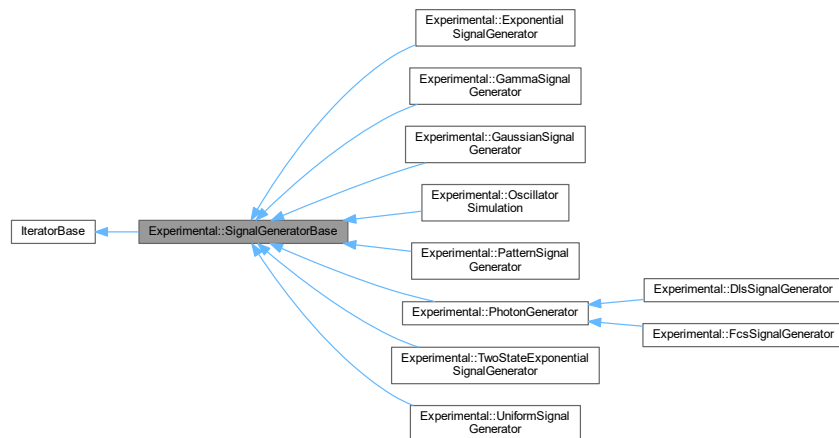
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.57 Experimental::SignalGeneratorBase Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::SignalGeneratorBase:



### Public Member Functions

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
the new virtual channel

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- virtual void [initialize](#) ([timestamp\\_t](#) [initial\\_time](#))=0
- virtual [timestamp\\_t](#) [get\\_next](#) ()=0
- virtual void [on\\_restart](#) ([timestamp\\_t](#) [restart\\_time](#))
- bool [next\\_impl](#) (std::vector< [Tag](#) > &[incoming\\_tags](#), [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#)) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool [is\\_finished](#))

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*[tagger](#), std::string [base\\_type](#)="IteratorBase", std::string [extra\\_info](#)="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) [channel](#))  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) [channel](#))  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()

- clear [Iterator](#) state.
- virtual void [on\\_start](#) ()  
callback when the measurement class is started
- void [lock](#) ()  
acquire update lock
- void [unlock](#) ()  
release update lock
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
release lock and continue work in parallel
- std::unique\_lock< std::mutex > [getLock](#) ()  
acquire update lock
- void [finish\\_running](#) ()  
Callback for the measurement to stop itself.
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

### Protected Attributes

- std::unique\_ptr< [SignalGeneratorBaseImpl](#) > impl

### Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
list of channels used by the iterator
- bool [running](#)  
running state of the iterator
- bool [autostart](#)  
Condition if this measurement shall be started by the finishInitialization callback.
- [TimeTaggerBase](#) \* [tagger](#)  
Pointer to the corresponding Time Tagger object.
- [timestamp\\_t](#) [capture\\_duration](#)  
Duration the iterator has already processed data.
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
For internal use.
- std::atomic< bool > [aborting](#)

## 9.57.1 Constructor & Destructor Documentation

### 9.57.1.1 [SignalGeneratorBase](#)()

```
Experimental::SignalGeneratorBase::SignalGeneratorBase (
    TimeTaggerBase * tagger,
    channel\_t base_channel = CHANNEL\_UNUSED )
```

### 9.57.1.2 [~SignalGeneratorBase](#)()

```
Experimental::SignalGeneratorBase::~~SignalGeneratorBase ( )
```

## 9.57.2 Member Function Documentation

### 9.57.2.1 `get_next()`

```
virtual timestamp_t Experimental::SignalGeneratorBase::get_next ( ) [protected], [pure virtual]
```

Implemented in [Experimental::PhotonGenerator](#), [Experimental::UniformSignalGenerator](#), [Experimental::GaussianSignalGenerator](#), [Experimental::OscillatorSimulation](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::GammaSignalGenerator](#), and [Experimental::PatternSignalGenerator](#).

### 9.57.2.2 `getChannel()`

```
channel_t Experimental::SignalGeneratorBase::getChannel ( )
```

the new virtual channel

This function returns the new allocated virtual channel. It can be used now in any new iterator.

### 9.57.2.3 `initialize()`

```
virtual void Experimental::SignalGeneratorBase::initialize (
    timestamp_t initial_time ) [protected], [pure virtual]
```

Implemented in [Experimental::PhotonGenerator](#), [Experimental::UniformSignalGenerator](#), [Experimental::GaussianSignalGenerator](#), [Experimental::OscillatorSimulation](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::GammaSignalGenerator](#), and [Experimental::PatternSignalGenerator](#).

### 9.57.2.4 `isProcessingFinished()`

```
bool Experimental::SignalGeneratorBase::isProcessingFinished ( ) [protected]
```

### 9.57.2.5 `next_impl()`

```
bool Experimental::SignalGeneratorBase::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.57.2.6 on\_restart()**

```
virtual void Experimental::SignalGeneratorBase::on_restart (
    timestamp_t restart_time ) [protected], [virtual]
```

Reimplemented in [Experimental::PhotonGenerator](#), [Experimental::UniformSignalGenerator](#), [Experimental::GaussianSignalGenerator](#), [Experimental::OscillatorSimulation](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::GammaSignalGenerator](#), and [Experimental::PatternSignalGenerator](#).

**9.57.2.7 on\_stop()**

```
void Experimental::SignalGeneratorBase::on_stop ( ) [override], [protected], [virtual]
```

callback when the measurement class is stopped

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.57.2.8 set\_processing\_finished()**

```
void Experimental::SignalGeneratorBase::set_processing_finished (
    bool is_finished ) [protected]
```

**9.57.3 Member Data Documentation****9.57.3.1 impl**

```
std::unique_ptr<SignalGeneratorBaseImpl> Experimental::SignalGeneratorBase::impl [protected]
```

The documentation for this class was generated from the following file:

- [Iterators.h](#)

**9.58 Experimental::SimDetector Class Reference**

```
#include <Iterators.h>
```

## Public Member Functions

- [SimDetector](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, double efficiency=1.0, double darkcount←\_rate=0.0, double jitter=0, double deadtime=0.0, int32\_t seed=-1)  
Construct a simulation of a physical detector for a given channel/signal.
- [~SimDetector](#) ()
- [channel\\_t](#) getChannel ()

## 9.58.1 Constructor & Destructor Documentation

### 9.58.1.1 SimDetector()

```
Experimental::SimDetector::SimDetector (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    double efficiency = 1.0,
    double darkcount_rate = 0.0,
    double jitter = 0,
    double deadtime = 0.0,
    int32_t seed = -1 )
```

Construct a simulation of a physical detector for a given channel/signal.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel with the signal passing through the virtual detector
<i>efficiency</i>	rate of acceptance for inputs.
<i>darkcount_rate</i>	rate of noise in Herz.
<i>jitter</i>	standard deviation of the gaussian broadening, in seconds.
<i>deadtime</i>	deadtime, in seconds.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.58.1.2 ~SimDetector()

```
Experimental::SimDetector::~SimDetector ( )
```

## 9.58.2 Member Function Documentation

### 9.58.2.1 getChannel()

```
channel_t Experimental::SimDetector::getChannel ( )
```

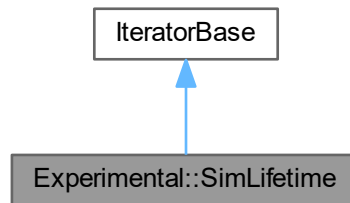
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.59 Experimental::SimLifetime Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::SimLifetime:



### Public Member Functions

- `SimLifetime` (`TimeTaggerBase` \*tagger, `channel_t` input\_channel, double lifetime, double emission\_rate=0.1, `int32_t` seed=-1)  
*Construct a simulation of a physical exaltation.*
- `~SimLifetime` ()
- `channel_t` getChannel ()
- void `registerLifetimeReactor` (`channel_t` trigger\_channel, `std::vector`< double > lifetimes, bool repeat)
- void `registerEmissionReactor` (`channel_t` trigger\_channel, `std::vector`< double > emissions, bool repeat)

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void `start` ()  
*Starts or continues data acquisition.*
- void `startFor` (`timestamp_t` capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool `waitUntilFinished` (`int64_t` timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- void `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` getCaptureDuration ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` getConfiguration ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`



## 9.59.1 Constructor & Destructor Documentation

### 9.59.1.1 SimLifetime()

```
Experimental::SimLifetime::SimLifetime (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    double lifetime,
    double emission_rate = 0.1,
    int32_t seed = -1 )
```

Construct a simulation of a physical exaltation.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel which triggers the exaltation.
<i>lifetime</i>	lifetime of the exaltation.
<i>emission_rate</i>	poissonian emission rate for each input event.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.59.1.2 ~SimLifetime()

```
Experimental::SimLifetime::~~SimLifetime ( )
```

## 9.59.2 Member Function Documentation

### 9.59.2.1 getChannel()

```
channel_t Experimental::SimLifetime::getChannel ( )
```

### 9.59.2.2 next\_impl()

```
bool Experimental::SimLifetime::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.59.2.3 registerEmissionReactor()**

```
void Experimental::SimLifetime::registerEmissionReactor (
    channel_t trigger_channel,
    std::vector< double > emissions,
    bool repeat )
```

**9.59.2.4 registerLifetimeReactor()**

```
void Experimental::SimLifetime::registerLifetimeReactor (
    channel_t trigger_channel,
    std::vector< double > lifetimes,
    bool repeat )
```

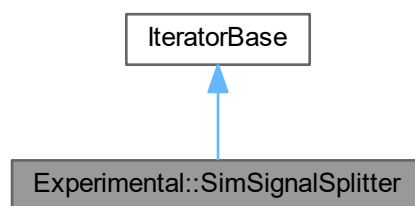
The documentation for this class was generated from the following file:

- [Iterators.h](#)

**9.60 Experimental::SimSignalSplitter Class Reference**

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::SimSignalSplitter:

**Public Member Functions**

- [SimSignalSplitter](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, double ratio=0.5, int32\_t seed=-1)  
Construct a signal splitter which will split events from an input channel into a left and a right virtual channels.
- [~SimSignalSplitter](#) ()
- std::vector< [channel\\_t](#) > [getChannels](#) ()
- [channel\\_t](#) [getLeftChannel](#) ()
- [channel\\_t](#) [getRightChannel](#) ()

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` [getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- `template<typename T >`  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t >` [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- `timestamp_t` [pre\\_capture\\_duration](#)  
*For internal use.*
- `std::atomic< bool >` [aborting](#)

## 9.60.1 Constructor & Destructor Documentation

### 9.60.1.1 SimSignalSplitter()

```
Experimental::SimSignalSplitter::SimSignalSplitter (
    TimeTaggerBase * tagger,
    channel\_t input_channel,
    double ratio = 0.5,
    int32_t seed = -1 )
```

Construct a signal splitter which will split events from an input channel into a left and a right virtual channels.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel to be split.
<i>ratio</i>	bias towards right or left channel.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.60.1.2 ~SimSignalSplitter()

```
Experimental::SimSignalSplitter::~~SimSignalSplitter ( )
```

## 9.60.2 Member Function Documentation

### 9.60.2.1 getChannels()

```
std::vector< channel_t > Experimental::SimSignalSplitter::getChannels ( )
```

### 9.60.2.2 getLeftChannel()

```
channel_t Experimental::SimSignalSplitter::getLeftChannel ( )
```

### 9.60.2.3 getRightChannel()

```
channel_t Experimental::SimSignalSplitter::getRightChannel ( )
```

### 9.60.2.4 next\_impl()

```
bool Experimental::SimSignalSplitter::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.61 SoftwareClockState Struct Reference

```
#include <TimeTagger.h>
```

### Public Attributes

- [timestamp\\_t](#) clock\_period
- [channel\\_t](#) input\_channel
- [channel\\_t](#) ideal\_clock\_channel
- [double](#) averaging\_periods
- [bool](#) enabled
- [bool](#) is\_locked
- [uint32\\_t](#) error\_counter
- [timestamp\\_t](#) last\_ideal\_clock\_event
- [double](#) period\_error
- [double](#) phase\_error\_estimation

### 9.61.1 Member Data Documentation

#### 9.61.1.1 averaging\_periods

```
double SoftwareClockState::averaging_periods
```

#### 9.61.1.2 clock\_period

```
timestamp_t SoftwareClockState::clock_period
```

#### 9.61.1.3 enabled

```
bool SoftwareClockState::enabled
```

#### 9.61.1.4 error\_counter

```
uint32_t SoftwareClockState::error_counter
```

#### 9.61.1.5 ideal\_clock\_channel

```
channel_t SoftwareClockState::ideal_clock_channel
```

#### 9.61.1.6 input\_channel

```
channel_t SoftwareClockState::input_channel
```

### 9.61.1.7 is\_locked

```
bool SoftwareClockState::is_locked
```

### 9.61.1.8 last\_ideal\_clock\_event

```
timestamp_t SoftwareClockState::last_ideal_clock_event
```

### 9.61.1.9 period\_error

```
double SoftwareClockState::period_error
```

### 9.61.1.10 phase\_error\_estimation

```
double SoftwareClockState::phase_error_estimation
```

The documentation for this struct was generated from the following file:

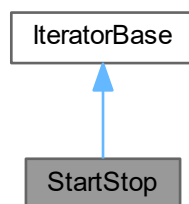
- [TimeTagger.h](#)

## 9.62 StartStop Class Reference

simple start-stop measurement

```
#include <Iterators.h>
```

Inheritance diagram for StartStop:



### Public Member Functions

- [StartStop](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) click\_channel, [channel\\_t](#) start\_channel=[CHANNEL\\_UNUSED](#), [timestamp\\_t](#) binwidth=1000)  
*constructor of StartStop*
- [~StartStop](#) ()
- void [getData](#) (std::function< [timestamp\\_t](#) \*(size\_t, size\_t)> array\_out)

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()



- acquire update lock*
- void `unlock()`
- release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` `getLock()`
- acquire update lock*
- void `finish_running()`
- Callback for the measurement to stop itself.*
- void `checkForAbort()`
- template<typename T>
- void `checkForAbort` (T callback)

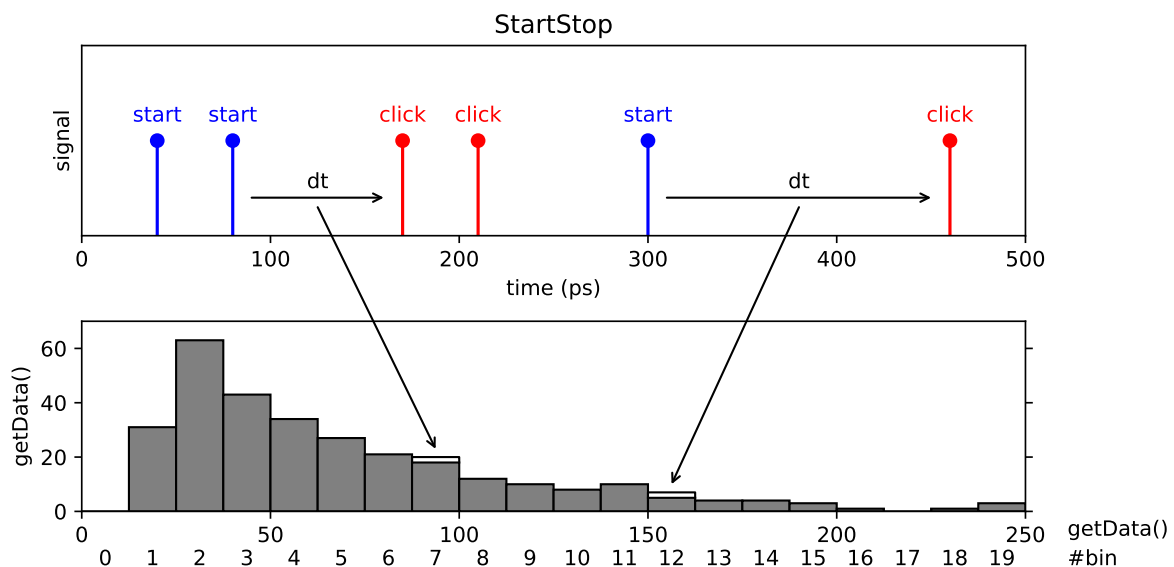
### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t >` `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase *` `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- `std::atomic< bool >` `aborting`

### 9.62.1 Detailed Description

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.

Be aware, on long-running measurements this may considerably slow down system performance and even crash the system entirely when attached to an unsuitable signal source.

## 9.62.2 Constructor & Destructor Documentation

### 9.62.2.1 StartStop()

```
StartStop::StartStop (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel = CHANNEL_UNUSED,
    timestamp_t binwidth = 1000 )
```

constructor of [StartStop](#)

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel for stop clicks
<i>start_channel</i>	channel for start clicks
<i>binwidth</i>	width of one histogram bin in ps

### 9.62.2.2 ~StartStop()

```
StartStop::~StartStop ( )
```

## 9.62.3 Member Function Documentation

### 9.62.3.1 clear\_impl()

```
void StartStop::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.62.3.2 getData()

```
void StartStop::getData (
    std::function< timestamp_t *(size_t, size_t)> array_out )
```

### 9.62.3.3 next\_impl()

```
bool StartStop::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.62.3.4 on\_start()

```
void StartStop::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.63 SynchronizedMeasurements Class Reference

start, stop and clear several measurements synchronized

```
#include <Iterators.h>
```

## Public Member Functions

- [SynchronizedMeasurements](#) ([TimeTaggerBase](#) \*tagger)  
*construct a [SynchronizedMeasurements](#) object*
- [~SynchronizedMeasurements](#) ()
- void [registerMeasurement](#) ([IteratorBase](#) \*measurement)  
*register a measurement (iterator) to the [SynchronizedMeasurements](#)-group.*
- void [unregisterMeasurement](#) ([IteratorBase](#) \*measurement)  
*unregister a measurement (iterator) from the [SynchronizedMeasurements](#)-group.*
- void [clear](#) ()  
*clear all registered measurements synchronously*
- void [start](#) ()  
*start all registered measurements synchronously*
- void [stop](#) ()  
*stop all registered measurements synchronously*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool [clear](#)=true)  
*start all registered measurements synchronously, and stops them after the capture\_duration*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*wait until all registered measurements have finished running.*
- bool [isRunning](#) ()  
*check if any iterator is running*
- [TimeTaggerBase](#) \* [getTagger](#) ()  
*Returns a proxy tagger object, which shall be used to create immediately registered measurements.*

## Protected Member Functions

- void [runCallback](#) ([TimeTaggerBase::IteratorCallback](#) callback, std::unique\_lock< std::mutex > &lk, bool block=true)  
*run a callback on all registered measurements synchronously*

### 9.63.1 Detailed Description

start, stop and clear several measurements synchronized

For the case that several measurements should be started, stopped or cleared at the very same time, a [SynchronizedMeasurements](#) object can be create to which all the measurements (also called iterators) can be registered with [.registerMeasurement\(measurement\)](#). Calling [.stop\(\)](#), [.start\(\)](#) or [.clear\(\)](#) on the [SynchronizedMeasurements](#) object will call the respective method on each of the registered measurements at the very same time. That means that all measurements taking part will have processed the very same time tags.

### 9.63.2 Constructor & Destructor Documentation

#### 9.63.2.1 SynchronizedMeasurements()

```
SynchronizedMeasurements::SynchronizedMeasurements (
    TimeTaggerBase * tagger )
```

construct a [SynchronizedMeasurements](#) object

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
---------------	---

**9.63.2.2 ~SynchronizedMeasurements()**

```
SynchronizedMeasurements::~~SynchronizedMeasurements ( )
```

**9.63.3 Member Function Documentation****9.63.3.1 clear()**

```
void SynchronizedMeasurements::clear ( )
```

clear all registered measurements synchronously

**9.63.3.2 getTagger()**

```
TimeTaggerBase * SynchronizedMeasurements::getTagger ( )
```

Returns a proxy tagger object, which shall be used to create immediately registered measurements.

Those measurements will not start automatically.

**9.63.3.3 isRunning()**

```
bool SynchronizedMeasurements::isRunning ( )
```

check if any iterator is running

**9.63.3.4 registerMeasurement()**

```
void SynchronizedMeasurements::registerMeasurement (
    IteratorBase * measurement )
```

register a measurement (iterator) to the SynchronizedMeasurements-group.

All available methods called on the [SynchronizedMeasurements](#) will happen at the very same time for all the registered measurements.

**9.63.3.5 runCallback()**

```
void SynchronizedMeasurements::runCallback (
    TimeTaggerBase::IteratorCallback callback,
    std::unique_lock< std::mutex > & lk,
    bool block = true ) [protected]
```

run a callback on all registered measurements synchronously

Please keep in mind that the callback is copied for each measurement. So please avoid big captures.

**9.63.3.6 start()**

```
void SynchronizedMeasurements::start ( )
```

start all registered measurements synchronously

**9.63.3.7 startFor()**

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

start all registered measurements synchronously, and stops them after the capture\_duration

**9.63.3.8 stop()**

```
void SynchronizedMeasurements::stop ( )
```

stop all registered measurements synchronously

**9.63.3.9 unregisterMeasurement()**

```
void SynchronizedMeasurements::unregisterMeasurement (
    IteratorBase * measurement )
```

unregister a measurement (iterator) from the SynchronizedMeasurements-group.

Stops synchronizing calls on the selected measurement, if the measurement is not within this synchronized group, the method does nothing.

**9.63.3.10 waitUntilFinished()**

```
bool SynchronizedMeasurements::waitUntilFinished (
    int64_t timeout = -1 )
```

wait until all registered measurements have finished running.

**Parameters**

<i>timeout</i>	time in milliseconds to wait for the measurements. If negative, wait until finished.
----------------	--

waitUntilFinished will wait according to the timeout and return true if all measurements finished or false if not. Furthermore, when waitUntilFinished is called on a set running indefinitely, it will log an error and return immediately.

The documentation for this class was generated from the following file:

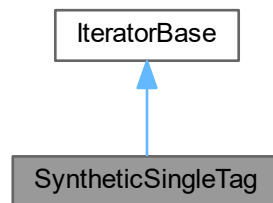
- [Iterators.h](#)

## 9.64 SyntheticSingleTag Class Reference

synthetic trigger timetag generator.

```
#include <Iterators.h>
```

Inheritance diagram for SyntheticSingleTag:



### Public Member Functions

- `SyntheticSingleTag` (`TimeTaggerBase` \*`tagger`, `channel_t` `base_channel`=`CHANNEL_UNUSED`)  
Construct a pulse event generator.
- `~SyntheticSingleTag` ()
- `void trigger` ()  
Generate a timetag for each call of this method.
- `channel_t getChannel` () const

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
destructor, will unregister from the Time Tagger prior finalization.
- `void start` ()  
Starts or continues data acquisition.
- `void startFor` (`timestamp_t` `capture_duration`, bool `clear`=true)  
Starts or continues the data acquisition for the given duration.
- bool `waitUntilFinished` (int64\_t `timeout`=-1)  
Blocks the execution until the measurement has finished. Can be used with `startFor()`.
- `void stop` ()  
After calling this method, the measurement will stop processing incoming tags.
- `void clear` ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- `void abort` ()  
Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.
- bool `isRunning` ()  
Returns True if the measurement is collecting the data.
- `timestamp_t getCaptureDuration` ()  
Total capture duration since the measurement creation or last call to `clear()`.
- `std::string getConfiguration` ()  
Fetches the overall configuration status of the measurement.

### Protected Member Functions

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`



### 9.64.1 Detailed Description

synthetic trigger timetag generator.

Creates timetags based on a trigger method. Whenever the user calls the 'trigger' method, a timetag will be added to the base\_channel.

This synthetic channel can inject timetags into an existing channel or create a new virtual channel.

### 9.64.2 Constructor & Destructor Documentation

#### 9.64.2.1 SyntheticSingleTag()

```
SyntheticSingleTag::SyntheticSingleTag (
    TimeTaggerBase * tagger,
    channel_t base_channel = CHANNEL_UNUSED )
```

Construct a pulse event generator.

##### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.

#### 9.64.2.2 ~SyntheticSingleTag()

```
SyntheticSingleTag::~SyntheticSingleTag ( )
```

### 9.64.3 Member Function Documentation

#### 9.64.3.1 getChannel()

```
channel_t SyntheticSingleTag::getChannel ( ) const
```

#### 9.64.3.2 next\_impl()

```
bool SyntheticSingleTag::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

**Parameters**

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.64.3.3 trigger()**

```
void SyntheticSingleTag::trigger ( )
```

Generate a timetag for each call of this method.

The documentation for this class was generated from the following file:

- [Iterators.h](#)

**9.65 Tag Struct Reference**

a single event on a channel

```
#include <TimeTagger.h>
```

**Public Types**

- enum class [Type](#) : unsigned char {  
[TimeTag](#) = 0 , [Error](#) = 1 , [OverflowBegin](#) = 2 , [OverflowEnd](#) = 3 ,  
[MissedEvents](#) = 4 }

*This enum marks what kind of event this object represents.*

**Public Member Functions**

- [Tag](#) ()
- [Tag](#) (timestamp\_t ts, channel\_t ch, [Type](#) type=[Type::TimeTag](#))
- [Tag](#) ([Type](#) type, char [reserved](#), unsigned short [missed\\_events](#), channel\_t ch, timestamp\_t ts)

## Public Attributes

- enum [Tag::Type](#) [TimeTag](#)
- char [reserved](#) {}  
*8 bit padding*
- unsigned short [missed\\_events](#) {}  
*Amount of missed events in overflow mode.*
- [channel\\_t](#) [channel](#) {}  
*the channel number*
- [timestamp\\_t](#) [time](#) {}  
*the timestamp of the event in picoseconds*

### 9.65.1 Detailed Description

a single event on a channel

Channel events are passed from the backend to registered iterators by the `IteratorBase::next()` callback function.

A [Tag](#) describes a single event on a channel.

### 9.65.2 Member Enumeration Documentation

#### 9.65.2.1 Type

```
enum class Tag::Type : unsigned char [strong]
```

This enum marks what kind of event this object represents.

- `TimeTag`: a normal event from any input channel
- `Error`: an error in the internal data processing, e.g. on plugging the external clock. This invalidates the global time
- `OverflowBegin`: this marks the begin of an interval with incomplete data because of too high data rates
- `OverflowEnd`: this marks the end of the interval. All events, which were lost in this interval, have been handled
- `MissedEvents`: this virtual event signals the amount of lost events per channel within an overflow interval. Repeated usage for higher amounts of events

#### Enumerator

TimeTag	
Error	
OverflowBegin	
OverflowEnd	
MissedEvents	

### 9.65.3 Constructor & Destructor Documentation

#### 9.65.3.1 Tag() [1/3]

```
Tag::Tag ( ) [inline]
```

#### 9.65.3.2 Tag() [2/3]

```
Tag::Tag (
    timestamp_t ts,
    channel_t ch,
    Type type = Type::TimeTag ) [inline]
```

#### 9.65.3.3 Tag() [3/3]

```
Tag::Tag (
    Type type,
    char reserved,
    unsigned short missed_events,
    channel_t ch,
    timestamp_t ts ) [inline]
```

### 9.65.4 Member Data Documentation

#### 9.65.4.1 channel

```
channel_t Tag::channel {}
```

the channel number

#### 9.65.4.2 missed\_events

```
unsigned short Tag::missed_events {}
```

Amount of missed events in overflow mode.

Within overflow intervals, the timing of all events is skipped. However, the total amount of events is still recorded. For events with type = MissedEvents, this indicates that a given amount of tags for this channel have been skipped in the interval. Note: There might be many missed events tags per overflow interval and channel. The accumulated amount represents the total skipped events.

#### 9.65.4.3 reserved

```
char Tag::reserved {}
```

8 bit padding

Reserved for future use. Set it to zero.

#### 9.65.4.4 time

```
timestamp_t Tag::time {}
```

the timestamp of the event in picoseconds

#### 9.65.4.5 TimeTag

```
enum Tag::Type Tag::TimeTag
```

The documentation for this struct was generated from the following file:

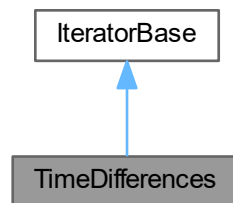
- [TimeTagger.h](#)

## 9.66 TimeDifferences Class Reference

Accumulates the time differences between clicks on two channels in one or more histograms.

```
#include <Iterators.h>
```

Inheritance diagram for TimeDifferences:



### Public Member Functions

- [TimeDifferences](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) click\_channel, [channel\\_t](#) start\_channel=[CHANNEL\\_UNUSED](#), [channel\\_t](#) next\_channel=[CHANNEL\\_UNUSED](#), [channel\\_t](#) sync\_channel=[CHANNEL\\_UNUSED](#), [timestamp\\_t](#) binwidth=1000, [int32\\_t](#) n\_bins=1000, [int32\\_t](#) n\_histograms=1)  
*constructor of a [TimeDifferences](#) measurement*
- [~TimeDifferences](#) ()
- void [getData](#) (std::function< [int32\\_t](#) \*([size\\_t](#), [size\\_t](#))> array\_out)  
*returns a two-dimensional array of size 'n\_bins' by 'n\_histograms' containing the histograms*
- void [getIndex](#) (std::function< [timestamp\\_t](#) \*([size\\_t](#))> array\_out)  
*returns a vector of size 'n\_bins' containing the time bins in ps*
- void [setMaxCounts](#) ([uint64\\_t](#) max\_counts)  
*set the number of rollovers at which the measurement stops integrating*
- [uint64\\_t](#) [getCounts](#) ()  
*returns the number of rollovers (histogram index resets)*
- [int32\\_t](#) [getHistogramIndex](#) () const  
*The index of the currently processed histogram or the waiting state.*
- bool [ready](#) ()  
*returns 'true' when the required number of rollovers set by 'setMaxCounts' has been reached*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*
- void [on\\_start](#) () override  
*callback when the measurement class is started*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void `unlock` ()
- release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)
- release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` `getLock` ()
- acquire update lock*
- void `finish_running` ()
- Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >
- void `checkForAbort` (T callback)

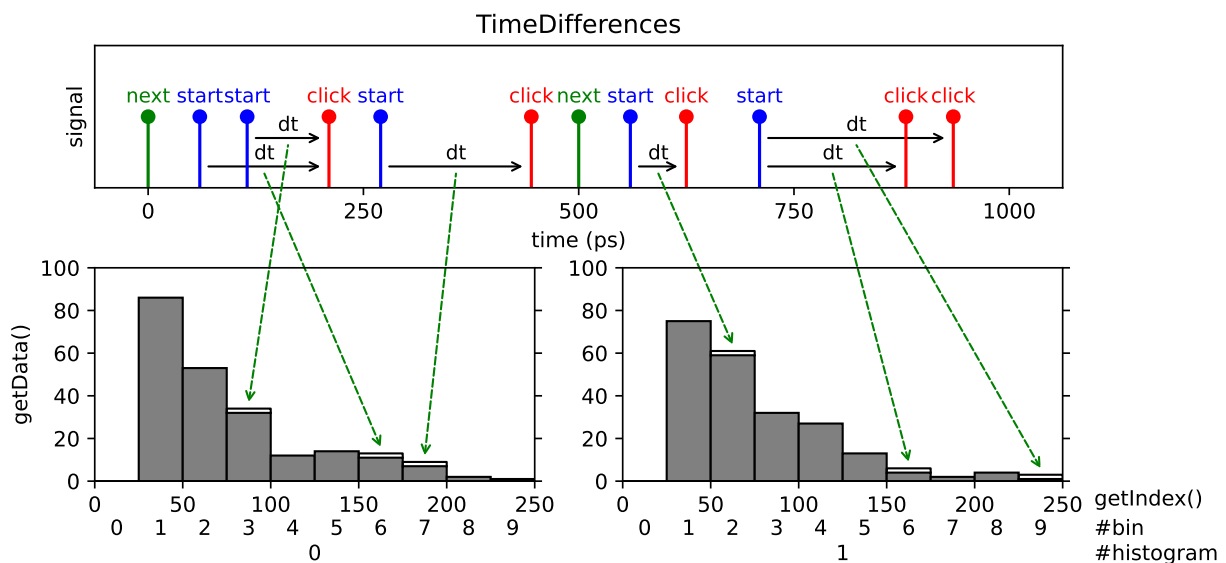
## Additional Inherited Members

## Protected Attributes inherited from `IteratorBase`

- `std::set< channel_t >` `channels_registered`
- list of channels used by the iterator*
- bool `running`
- running state of the iterator*
- bool `autostart`
- Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase *` `tagger`
- Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`
- Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`
- For internal use.*
- `std::atomic< bool >` `aborting`

### 9.66.1 Detailed Description

Accumulates the time differences between clicks on two channels in one or more histograms.



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

The data obtained from subsequent start tags can be accumulated into the same histogram (one-dimensional measurement) or into different histograms (two-dimensional measurement). In this way, you can perform more general two-dimensional time-difference measurements. The parameter 'n\_histograms' specifies the number of histograms. After each tag on the 'next\_channel', the histogram index is incremented by one and reset to zero after reaching the last valid index. The measurement starts with the first tag on the 'next\_channel'.

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

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

## 9.66.2 Constructor & Destructor Documentation

### 9.66.2.1 TimeDifferences()

```
TimeDifferences::TimeDifferences (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel = CHANNEL_UNUSED,
    channel_t next_channel = CHANNEL_UNUSED,
    channel_t sync_channel = CHANNEL_UNUSED,
    timestamp_t binwidth = 1000,
    int32_t n_bins = 1000,
    int32_t n_histograms = 1 )
```

constructor of a [TimeDifferences](#) measurement

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel that increments the count in a bin
<i>start_channel</i>	channel that sets start times relative to which clicks on the click channel are measured
<i>next_channel</i>	channel that increments the histogram index
<i>sync_channel</i>	channel that resets the histogram index to zero
<i>binwidth</i>	width of one histogram bin in ps
<i>n_bins</i>	number of bins in each histogram
<i>n_histograms</i>	number of histograms



### 9.66.2.2 ~TimeDifferences()

```
TimeDifferences::~~TimeDifferences ( )
```

## 9.66.3 Member Function Documentation

### 9.66.3.1 clear\_impl()

```
void TimeDifferences::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.66.3.2 getCounts()

```
uint64_t TimeDifferences::getCounts ( )
```

returns the number of rollovers (histogram index resets)

### 9.66.3.3 getData()

```
void TimeDifferences::getData (
    std::function< int32_t *(size_t, size_t)> array_out )
```

returns a two-dimensional array of size 'n\_bins' by 'n\_histograms' containing the histograms

### 9.66.3.4 getHistogramIndex()

```
int32_t TimeDifferences::getHistogramIndex ( ) const
```

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

### 9.66.3.5 getIndex()

```
void TimeDifferences::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

returns a vector of size 'n\_bins' containing the time bins in ps

### 9.66.3.6 next\_impl()

```
bool TimeDifferences::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.66.3.7 on\_start()**

```
void TimeDifferences::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.66.3.8 ready()**

```
bool TimeDifferences::ready ( )
```

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

**9.66.3.9 setMaxCounts()**

```
void TimeDifferences::setMaxCounts (
    uint64_t max_counts )
```

set the number of rollovers at which the measurement stops integrating

## Parameters

<i>max_counts</i>	maximum number of sync/next clicks
-------------------	------------------------------------

The documentation for this class was generated from the following file:

- [Iterators.h](#)

**9.67 TimeDifferencesImpl< T > Class Template Reference**

The documentation for this class was generated from the following file:

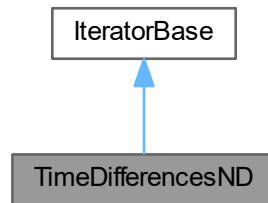
- [Iterators.h](#)

## 9.68 TimeDifferencesND Class Reference

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

```
#include <Iterators.h>
```

Inheritance diagram for TimeDifferencesND:



### Public Member Functions

- `TimeDifferencesND` (`TimeTaggerBase` \*tagger, `channel_t` click\_channel, `channel_t` start\_channel, `std::vector`< `channel_t` > next\_channels, `std::vector`< `channel_t` > sync\_channels, `std::vector`< `int32_t` > n\_hists, `timestamp_t` binwidth, `int32_t` n\_bins)  
*constructor of a `TimeDifferencesND` measurement*
- `~TimeDifferencesND` ()
- `void` `getData` (`std::function`< `int32_t` \*(`size_t`, `size_t`)> array\_out)  
*returns a two-dimensional array of size `n_bins` by all `n_hists` containing the histograms*
- `void` `getIndex` (`std::function`< `timestamp_t` \*(`size_t`)> array\_out)  
*returns a vector of size `n_bins` containing the time bins in ps*

### Public Member Functions inherited from `IteratorBase`

- `virtual` `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- `void` `start` ()  
*Starts or continues data acquisition.*
- `void` `startFor` (`timestamp_t` capture\_duration, `bool` clear=true)  
*Starts or continues the data acquisition for the given duration.*
- `bool` `waitUntilFinished` (`int64_t` timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- `void` `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- `void` `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- `void` `abort` ()

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

- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t` `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string` `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (`std::vector`< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*
- void `clear_impl` () override  
*clear `Iterator` state.*
- void `on_start` () override  
*callback when the measurement class is started*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, `std::string` base\_type\_="IteratorBase", `std::string` extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock`< `std::mutex` > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- `template`<typename T >  
void `checkForAbort` (T callback)

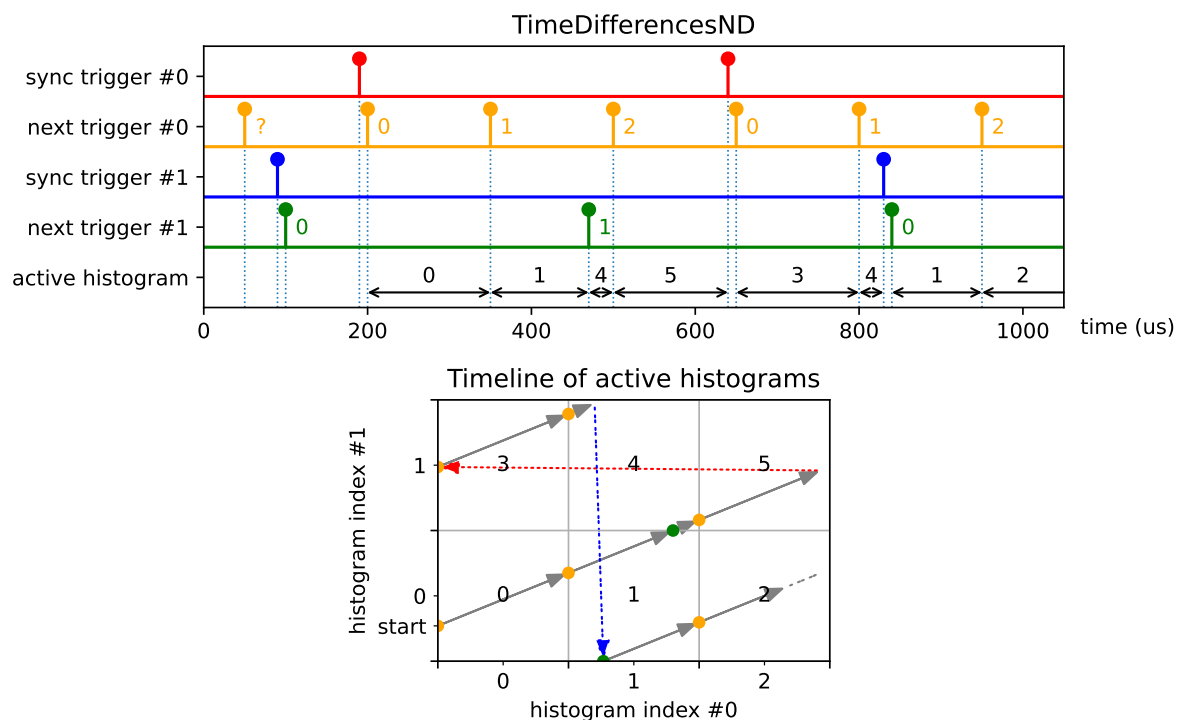
## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t pre_capture_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

### 9.68.1 Detailed Description

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



This is a multidimensional implementation of the [TimeDifferences](#) measurement class. Please read their documentation first.

This measurement class extends the [TimeDifferences](#) interface for a multidimensional amount of histograms. It captures many multiple start - multiple stop histograms, but with many asynchronous next\_channel triggers. After

each tag on each `next_channel`, the histogram index of the associated dimension is incremented by one and reset to zero after reaching the last valid index. The elements of the parameter `n_histograms` specifies the number of histograms per dimension. The accumulation starts when `next_channel` has been triggered on all dimensions.

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

Please use `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

## 9.68.2 Constructor & Destructor Documentation

### 9.68.2.1 TimeDifferencesND()

```
TimeDifferencesND::TimeDifferencesND (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel,
    std::vector< channel_t > next_channels,
    std::vector< channel_t > sync_channels,
    std::vector< int32_t > n_histograms,
    timestamp_t binwidth,
    int32_t n_bins )
```

constructor of a [TimeDifferencesND](#) measurement

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>click_channel</i>	channel that increments the count in a bin
<i>start_channel</i>	channel that sets start times relative to which clicks on the click channel are measured
<i>next_channels</i>	vector of channels that increments the histogram index
<i>sync_channels</i>	vector of channels that resets the histogram index to zero
<i>n_histograms</i>	vector of numbers of histograms per dimension.
<i>binwidth</i>	width of one histogram bin in ps
<i>n_bins</i>	number of bins in each histogram

### 9.68.2.2 ~TimeDifferencesND()

```
TimeDifferencesND::~TimeDifferencesND ( )
```

## 9.68.3 Member Function Documentation

### 9.68.3.1 clear\_impl()

```
void TimeDifferencesND::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

### 9.68.3.2 getData()

```
void TimeDifferencesND::getData (
    std::function< int32_t *(size_t, size_t)> array_out )
```

returns a two-dimensional array of size `n_bins` by all `n_histograms` containing the histograms

### 9.68.3.3 getIndex()

```
void TimeDifferencesND::getIndex (
    std::function< timestamp_t *(size_t)> array_out )
```

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

### 9.68.3.4 next\_impl()

```
bool TimeDifferencesND::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	<i>begin_time</i> of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

### 9.68.3.5 on\_start()

```
void TimeDifferencesND::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

The documentation for this class was generated from the following file:

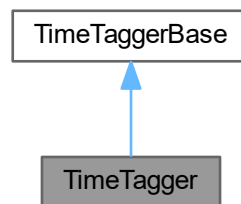
- [Iterators.h](#)

## 9.69 TimeTagger Class Reference

backend for the [TimeTagger](#).

```
#include <TimeTagger.h>
```

Inheritance diagram for TimeTagger:



### Public Member Functions

- virtual void [reset](#) ()=0  
*reset the [TimeTagger](#) object to default settings and detach all iterators*
- virtual bool [isChannelRegistered](#) ([channel\\_t](#) chan)=0
- virtual void [setTestSignalDivider](#) (int divider)=0  
*set the divider for the frequency of the test signal*
- virtual int [getTestSignalDivider](#) ()=0  
*get the divider for the frequency of the test signal*
- virtual void [xtra\\_setAuxOutSignal](#) (int channel, int divider, double duty\_cycle=0.5)=0  
*set the divider for the frequency of the aux out signal generator and enable aux out*
- virtual int [xtra\\_getAuxOutSignalDivider](#) (int channel)=0  
*get the divider for the frequency of the aux out signal generator*
- virtual double [xtra\\_getAuxOutSignalDutyCycle](#) (int channel)=0  
*get the dutycycle of the aux out signal generator*
- virtual void [xtra\\_setAuxOut](#) (int channel, bool enabled)=0  
*enable or disable aux out*
- virtual bool [xtra\\_getAuxOut](#) (int channel)=0  
*fetch the status of the aux out signal generator*



- virtual void [xtra\\_setFanSpeed](#) (double percentage=-1)=0  
*configures the FAN speed on TTU HW >= 1.3*
- virtual void [setTriggerLevel](#) ([channel\\_t](#) channel, double voltage)=0  
*set the trigger voltage threshold of a channel*
- virtual double [getTriggerLevel](#) ([channel\\_t](#) channel)=0  
*get the trigger voltage threshold of a channel*
- virtual double [xtra\\_measureTriggerLevel](#) ([channel\\_t](#) channel)=0  
*measures the electrically applied the trigger voltage threshold of a channel*
- virtual [timestamp\\_t](#) [getHardwareDelayCompensation](#) ([channel\\_t](#) channel)=0  
*get hardware delay compensation of a channel*
- virtual void [setInputMux](#) ([channel\\_t](#) channel, int mux\_mode)=0  
*configures the input multiplexer*
- virtual int [getInputMux](#) ([channel\\_t](#) channel)=0  
*fetches the configuration of the input multiplexer*
- virtual void [setConditionalFilter](#) (std::vector< [channel\\_t](#) > trigger, std::vector< [channel\\_t](#) > filtered, bool hardwareDelayCompensation=true)=0  
*configures the conditional filter*
- virtual void [clearConditionalFilter](#) ()=0  
*deactivates the conditional filter*
- virtual std::vector< [channel\\_t](#) > [getConditionalFilterTrigger](#) ()=0  
*fetches the configuration of the conditional filter*
- virtual std::vector< [channel\\_t](#) > [getConditionalFilterFiltered](#) ()=0  
*fetches the configuration of the conditional filter*
- virtual void [setNormalization](#) (std::vector< [channel\\_t](#) > channels, bool state)=0  
*enables or disables the normalization of the distribution.*
- virtual bool [getNormalization](#) ([channel\\_t](#) channel)=0  
*returns the the normalization of the distribution.*
- virtual void [setHardwareBufferSize](#) (int size)=0  
*sets the maximum USB buffer size*
- virtual int [getHardwareBufferSize](#) ()=0  
*queries the size of the USB queue*
- virtual void [setStreamBlockSize](#) (int max\_events, int max\_latency)=0  
*sets the maximum events and latency for the stream block size*
- virtual int [getStreamBlockSizeEvents](#) ()=0
- virtual int [getStreamBlockSizeLatency](#) ()=0
- virtual void [setEventDivider](#) ([channel\\_t](#) channel, unsigned int divider)=0  
*Divides the amount of transmitted edge per channel.*
- virtual unsigned int [getEventDivider](#) ([channel\\_t](#) channel)=0  
*Returns the factor of the dividing filter.*
- virtual void [autoCalibration](#) (std::function< double \*(size\_t)> array\_out)=0  
*runs a calibrations based on the on-chip uncorrelated signal generator.*
- virtual std::string [getSerial](#) ()=0  
*identifies the hardware by serial number*
- virtual std::string [getModel](#) ()=0  
*identifies the hardware by Time Tagger Model*
- virtual int [getChannelNumberScheme](#) ()=0  
*Fetch the configured numbering scheme for this [TimeTagger](#) object.*
- virtual std::vector< double > [getDACRange](#) ()=0  
*returns the minimum and the maximum voltage of the DACs as a trigger reference*
- virtual void [getDistributionCount](#) (std::function< uint64\_t \*(size\_t, size\_t)> array\_out)=0  
*get internal calibration data*

- virtual void [getDistributionPSecs](#) (std::function< double \*(size\_t, size\_t)> array\_out)=0  
*get internal calibration data*
- virtual std::vector< [channel\\_t](#) > [getChannelList](#) ([ChannelEdge](#) type=[ChannelEdge::All](#))=0  
*fetch a vector of all physical input channel ids*
- virtual [timestamp\\_t](#) [getPsPerClock](#) ()=0  
*fetch the duration of each clock cycle in picoseconds*
- virtual std::string [getPcbVersion](#) ()=0  
*Return the hardware version of the PCB board. Version 0 is everything before mid 2018 and with the channel configuration ZERO. version >= 1 is channel configuration ONE.*
- virtual std::string [getFirmwareVersion](#) ()=0  
*Return an unique identifier for the applied firmware.*
- virtual void [xtra\\_setClockSource](#) (int source)=0  
*manually overwrite the reference clock source*
- virtual int [xtra\\_getClockSource](#) ()=0  
*fetch the overwritten reference clock source*
- virtual void [xtra\\_setClockAutoSelect](#) (bool enabled)=0  
*activates auto clocking function*
- virtual bool [xtra\\_getClockAutoSelect](#) ()=0  
*queries if the auto clocking function is enabled*
- virtual void [xtra\\_setClockOut](#) (bool enabled)=0  
*enables the clock output*
- virtual std::string [getSensorData](#) ()=0  
*Show the status of the sensor data from the FPGA and peripherals on the console.*
- virtual void [setLED](#) (uint32\_t bitmask)=0  
*Enforce a state to the LEDs 0: led\_status[R] 16: led\_status[R] - mux 1: led\_status[G] 17: led\_status[G] - mux 2: led\_status[B] 18: led\_status[B] - mux 3: led\_power[R] 19: led\_power[R] - mux 4: led\_power[G] 20: led\_power[G] - mux 5: led\_power[B] 21: led\_power[B] - mux 6: led\_clock[R] 22: led\_clock[R] - mux 7: led\_clock[G] 23: led\_clock[G] - mux 8: led\_clock[B] 24: led\_clock[B] - mux.*
- virtual void [disableLEDs](#) (bool disabled)=0  
*disables the LEDs on the TT*
- virtual std::string [getDeviceLicense](#) ()=0  
*gets the license, installed on this device currently*
- virtual uint32\_t [factoryAccess](#) (uint32\_t pw, uint32\_t addr, uint32\_t data, uint32\_t mask, bool use\_wb=false)=0  
*Direct read/write access to WireIn/WireOuts in FPGA (mask==0 for readonly)*
- virtual void [setSoundFrequency](#) (uint32\_t freq\_hz)=0  
*Set the Time Taggers internal buzzer to a frequency in Hz (freq\_hz==0 to disable)*
- virtual void [enableFpgaLink](#) (std::vector< [channel\\_t](#) > channels, std::string destination\_mac, [FpgaLinkInterface](#) link\_interface=[FpgaLinkInterface::SFPP\\_10GE](#), bool exclusive=false)=0  
*Enable the FPGA link of the Time Tagger X.*
- virtual void [disableFpgaLink](#) ()=0  
*Disable the FPGA link of the Time Tagger X.*
- virtual void [startServer](#) ([AccessMode](#) access\_mode, std::vector< [channel\\_t](#) > channels=std::vector< [channel\\_t](#) >(), uint32\_t port=41101)=0  
*starts the Time Tagger server that will stream the time tags to the client.*
- virtual bool [isServerRunning](#) ()=0  
*check if the server is still running.*
- virtual void [stopServer](#) ()=0  
*stops the time tagger server if currently running, otherwise does nothing.*
- virtual void [setTimeTaggerNetworkStreamCompression](#) (bool active)=0  
*enable or disable additional compression of the timetag stream as sent over the network.*
- virtual void [setInputImpedanceHigh](#) ([channel\\_t](#) channel, bool high\_impedance)=0  
*enable high impedance termination mode*

- virtual bool [getInputImpedanceHigh](#) ([channel\\_t](#) channel)=0  
*query the state of the high impedance termination mode*
- virtual void [setInputHysteresis](#) ([channel\\_t](#) channel, int value)=0  
*configure the hysteresis voltage of the input comparator*
- virtual int [getInputHysteresis](#) ([channel\\_t](#) channel)=0  
*query the hysteresis voltage of the input comparator*
- virtual void [xtra\\_setAvgRisingFalling](#) ([channel\\_t](#) channel, bool enable)=0  
*configures if the rising and falling events shall be averaged*
- virtual bool [xtra\\_getAvgRisingFalling](#) ([channel\\_t](#) channel)=0  
*query if the rising and falling events shall be averaged*
- virtual void [xtra\\_setHighPrioChannel](#) ([channel\\_t](#) channel, bool enable)=0  
*configures if this channel shall exit overflow regions.*
- virtual bool [xtra\\_getHighPrioChannel](#) ([channel\\_t](#) channel)=0  
*if this channel shall exit overflow regions*
- virtual void [updateBMCFirmware](#) (const std::string &firmware)=0  
*updates the firmware of the Time Tagger X board management controller*

## Public Member Functions inherited from [TimeTaggerBase](#)

- virtual unsigned int [getFence](#) (bool alloc\_fence=true)=0  
*Generate a new fence object, which validates the current configuration and the current time.*
- virtual bool [waitForFence](#) (unsigned int fence, int64\_t timeout=-1)=0  
*Wait for a fence in the data stream.*
- virtual bool [sync](#) (int64\_t timeout=-1)=0  
*Sync the timetagger pipeline, so that all started iterators and their enabled channels are ready.*
- virtual [channel\\_t](#) [getInvertedChannel](#) ([channel\\_t](#) channel)=0  
*get the falling channel id for a rising channel and vice versa*
- virtual bool [isUnusedChannel](#) ([channel\\_t](#) channel)=0  
*compares the provided channel with CHANNEL\_UNUSED*
- virtual void [runSynchronized](#) (const [IteratorCallbackMap](#) &callbacks, bool block=true)=0  
*Run synchronized callbacks for a list of iterators.*
- virtual std::string [getConfiguration](#) ()=0  
*Fetches the overall configuration status of the Time Tagger object.*
- virtual void [setInputDelay](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void [setDelayHardware](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void [setDelaySoftware](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual [timestamp\\_t](#) [getInputDelay](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [getDelaySoftware](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [getDelayHardware](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [setDeadtime](#) ([channel\\_t](#) channel, [timestamp\\_t](#) deadtime)=0  
*set the deadtime between two edges on the same channel.*
- virtual [timestamp\\_t](#) [getDeadtime](#) ([channel\\_t](#) channel)=0  
*get the deadtime between two edges on the same channel.*
- virtual void [setTestSignal](#) ([channel\\_t](#) channel, bool enabled)=0

- enable/disable internal test signal on a channel.*
  - virtual void [setTestSignal](#) (std::vector< [channel\\_t](#) > channel, bool enabled)=0
- enable/disable internal test signal on multiple channels.*
  - virtual bool [getTestSignal](#) ([channel\\_t](#) channel)=0
- fetch the status of the test signal generator*
  - virtual void [setSoftwareClock](#) ([channel\\_t](#) input\_channel, double input\_frequency=10e6, double averaging\_↔ periods=1000, bool wait\_until\_locked=true)=0
- enables a software PLL to lock the time to an external clock*
  - virtual void [disableSoftwareClock](#) ()=0
- disabled the software PLL*
  - virtual [SoftwareClockState](#) [getSoftwareClockState](#) ()=0
- queries all state information of the software clock*
  - virtual long long [getOverflows](#) ()=0
- get overflow count*
  - virtual void [clearOverflows](#) ()=0
- clear overflow counter*
  - virtual long long [getOverflowsAndClear](#) ()=0
- get and clear overflow counter*

### Additional Inherited Members

### Public Types inherited from [TimeTaggerBase](#)

- typedef std::function< void([IteratorBase](#) \*) [IteratorCallback](#)>
- typedef std::map< [IteratorBase](#) \*, [IteratorCallback](#) > [IteratorCallbackMap](#)

### Protected Member Functions inherited from [TimeTaggerBase](#)

- [TimeTaggerBase](#) ()
- abstract interface class*
- virtual [~TimeTaggerBase](#) ()
- destructor*
- [TimeTaggerBase](#) (const [TimeTaggerBase](#) &)=delete
- [TimeTaggerBase](#) & operator= (const [TimeTaggerBase](#) &)=delete
- virtual std::shared\_ptr< [IteratorBaseListNode](#) > [addIterator](#) ([IteratorBase](#) \*it)=0
- virtual void [freeIterator](#) ([IteratorBase](#) \*it)=0
- virtual [channel\\_t](#) [getNewVirtualChannel](#) ()=0
- virtual void [freeVirtualChannel](#) ([channel\\_t](#) channel)=0
- virtual void [registerChannel](#) ([channel\\_t](#) channel)=0
- register a FPGA channel.*
- virtual void [registerChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [unregisterChannel](#) ([channel\\_t](#) channel)=0
- release a previously registered channel.*
- virtual void [unregisterChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [addChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [removeChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [release](#) ()=0

### 9.69.1 Detailed Description

backend for the [TimeTagger](#).

The [TimeTagger](#) class connects to the hardware, and handles the communication over the usb. There may be only one instance of the backend per physical device.

### 9.69.2 Member Function Documentation

#### 9.69.2.1 autoCalibration()

```
virtual void TimeTagger::autoCalibration (
    std::function< double *(size_t)> array_out ) [pure virtual]
```

runs a calibrations based on the on-chip uncorrelated signal generator.

#### 9.69.2.2 clearConditionalFilter()

```
virtual void TimeTagger::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to setConditionalFilter({},{})

#### 9.69.2.3 disableFpgaLink()

```
virtual void TimeTagger::disableFpgaLink ( ) [pure virtual]
```

Disable the FPGA link of the Time Tagger X.

#### 9.69.2.4 disableLEDs()

```
virtual void TimeTagger::disableLEDs (
    bool disabled ) [pure virtual]
```

disables the LEDs on the TT

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

##### Parameters

<i>disabled</i>	true to disable all LEDs on the TT
-----------------	------------------------------------

#### 9.69.2.5 enableFpgaLink()

```
virtual void TimeTagger::enableFpgaLink (
    std::vector< channel\_t > channels,
```

```
std::string destination_mac,
FpgaLinkInterface link_interface = FpgaLinkInterface::SFPP_10GE,
bool exclusive = false ) [pure virtual]
```

Enable the FPGA link of the Time Tagger X.

#### Parameters

<i>channels</i>	list of channels, which shall be streamed over the FPGA link
<i>destination_mac</i>	Destination MAC, use an empty string for the broadcast address of "FF:FF:FF:FF:FF:FF"
<i>link_interface</i>	selects which interface shall be used, default is <a href="#">FpgaLinkInterface::SFPP_10GE</a>
<i>exclusive</i>	determines if time tags should exclusively be transmitted over Ethernet, increasing Ethernet performance and avoiding USB issues, default is mixed USB & ethernet

#### 9.69.2.6 factoryAccess()

```
virtual uint32_t TimeTagger::factoryAccess (
    uint32_t pw,
    uint32_t addr,
    uint32_t data,
    uint32_t mask,
    bool use_wb = false ) [pure virtual]
```

Direct read/write access to WireIn/WireOuts in FPGA (mask==0 for readonly)

DO NOT USE. Only for internal debug purposes.

#### 9.69.2.7 getChannelList()

```
virtual std::vector< channel_t > TimeTagger::getChannelList (
    ChannelEdge type = ChannelEdge::All ) [pure virtual]
```

fetch a vector of all physical input channel ids

The function returns the channel of all rising and falling edges. For example for the Time Tagger 20 (8 input channels) TT\_CHANNEL\_NUMBER\_SCHEME\_ZERO: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15} and for TT\_CHANNEL\_NUMBER\_SCHEME\_ONE: {-8,-7,-6,-5,-4,-3,-2,-1,1,2,3,4,5,6,7,8}

TT\_CHANNEL\_RISING\_EDGES returns only the rising edges SCHEME\_ONE: {1,2,3,4,5,6,7,8} and TT\_CHANNEL\_FALLING\_EDGES return only the falling edges SCHEME\_ONE: {-1,-2,-3,-4,-5,-6,-7,-8} which are the invertedChannels of the rising edges.

#### 9.69.2.8 getChannelNumberScheme()

```
virtual int TimeTagger::getChannelNumberScheme ( ) [pure virtual]
```

Fetch the configured numbering scheme for this [TimeTagger](#) object.

Please see [setTimeTaggerChannelNumberScheme\(\)](#) for details.

#### 9.69.2.9 getConditionalFilterFiltered()

```
virtual std::vector< channel_t > TimeTagger::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

#### 9.69.2.10 getConditionalFilterTrigger()

```
virtual std::vector< channel_t > TimeTagger::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

#### 9.69.2.11 getDACRange()

```
virtual std::vector< double > TimeTagger::getDACRange ( ) [pure virtual]
```

returns the minimum and the maximum voltage of the DACs as a trigger reference

#### 9.69.2.12 getDeviceLicense()

```
virtual std::string TimeTagger::getDeviceLicense ( ) [pure virtual]
```

gets the license, installed on this device currently

##### Returns

a JSON string containing the current device license

#### 9.69.2.13 getDistributionCount()

```
virtual void TimeTagger::getDistributionCount (
    std::function< uint64_t *(size_t, size_t)> array_out ) [pure virtual]
```

get internal calibration data

#### 9.69.2.14 getDistributionPSecs()

```
virtual void TimeTagger::getDistributionPSecs (
    std::function< double *(size_t, size_t)> array_out ) [pure virtual]
```

get internal calibration data

#### 9.69.2.15 getEventDivider()

```
virtual unsigned int TimeTagger::getEventDivider (
    channel_t channel ) [pure virtual]
```

Returns the factor of the dividing filter.

See setEventDivider for further details.

**Parameters**

<i>channel</i>	channel to be queried
----------------	-----------------------

**Returns**

the configured divider

**9.69.2.16 getFirmwareVersion()**

```
virtual std::string TimeTagger::getFirmwareVersion ( ) [pure virtual]
```

Return an unique identifier for the applied firmware.

This function returns a comma separated list of the firmware version with

- the device identifier: TT-20 or TT-Ultra
- the firmware identifier: FW 3
- optional the timestamp of the assembling of the firmware
- the firmware identifier of the USB chip: OK 1.30 eg "TT-Ultra, FW 3, TS 2018-11-13 22:57:32, OK 1.30"

**9.69.2.17 getHardwareBufferSize()**

```
virtual int TimeTagger::getHardwareBufferSize ( ) [pure virtual]
```

queries the size of the USB queue

See setHardwareBufferSize for more information.

**Returns**

the actual size of the USB queue in events

**9.69.2.18 getHardwareDelayCompensation()**

```
virtual timestamp_t TimeTagger::getHardwareDelayCompensation (
    channel_t channel ) [pure virtual]
```

get 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

<i>channel</i>	the channel
----------------	-------------

## Returns

the hardware delay compensation in picoseconds

**9.69.2.19 getInputHysteresis()**

```
virtual int TimeTagger::getInputHysteresis (  
    channel_t channel ) [pure virtual]
```

query the hysteresis voltage of the input comparator

## Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

## Returns

the hysteresis voltage in milli Volt

**9.69.2.20 getInputImpedanceHigh()**

```
virtual bool TimeTagger::getInputImpedanceHigh (  
    channel_t channel ) [pure virtual]
```

query the state of the high impedance termination mode

## Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

## Returns

true for the high impedance mode or false for the 50 Ohm termination mode

**9.69.2.21 getInputMux()**

```
virtual int TimeTagger::getInputMux (  
    channel_t channel ) [pure virtual]
```

fetches the configuration of the input multiplexer

**Parameters**

<i>channel</i>	the physical channel of the input multiplexer
----------------	---

**Returns**

the configuration mode of the input multiplexer

**9.69.2.22 getModel()**

```
virtual std::string TimeTagger::getModel ( ) [pure virtual]
```

identifies the hardware by Time Tagger Model

**9.69.2.23 getNormalization()**

```
virtual bool TimeTagger::getNormalization (
    channel_t channel ) [pure virtual]
```

returns the the normalization of the distribution.

Refer the Manual for a description of this function.

**Parameters**

<i>channel</i>	the channel to query
----------------	----------------------

**Returns**

if the normalization is enabled

**9.69.2.24 getPcbVersion()**

```
virtual std::string TimeTagger::getPcbVersion ( ) [pure virtual]
```

Return the hardware version of the PCB board. Version 0 is everything before mid 2018 and with the channel configuration ZERO. version  $\geq 1$  is channel configuration ONE.

**9.69.2.25 getPsPerClock()**

```
virtual timestamp_t TimeTagger::getPsPerClock ( ) [pure virtual]
```

fetch the duration of each clock cycle in picoseconds

### 9.69.2.26 getSensorData()

```
virtual std::string TimeTagger::getSensorData ( ) [pure virtual]
```

Show the status of the sensor data from the FPGA and peripherals on the console.

### 9.69.2.27 getSerial()

```
virtual std::string TimeTagger::getSerial ( ) [pure virtual]
```

identifies the hardware by serial number

### 9.69.2.28 getStreamBlockSizeEvents()

```
virtual int TimeTagger::getStreamBlockSizeEvents ( ) [pure virtual]
```

### 9.69.2.29 getStreamBlockSizeLatency()

```
virtual int TimeTagger::getStreamBlockSizeLatency ( ) [pure virtual]
```

### 9.69.2.30 getTestSignalDivider()

```
virtual int TimeTagger::getTestSignalDivider ( ) [pure virtual]
```

get the divider for the frequency of the test signal

### 9.69.2.31 getTriggerLevel()

```
virtual double TimeTagger::getTriggerLevel (
    channel_t channel ) [pure virtual]
```

get the trigger voltage threshold of a channel

#### Parameters

<i>channel</i>	the channel
----------------	-------------

### 9.69.2.32 isChannelRegistered()

```
virtual bool TimeTagger::isChannelRegistered (
    channel_t chan ) [pure virtual]
```

### 9.69.2.33 isServerRunning()

```
virtual bool TimeTagger::isServerRunning ( ) [pure virtual]
```

check if the server is still running.

#### Returns

returns true if running; false, if not running

#### 9.69.2.34 reset()

```
virtual void TimeTagger::reset ( ) [pure virtual]
```

reset the [TimeTagger](#) object to default settings and detach all iterators

#### 9.69.2.35 setConditionalFilter()

```
virtual void TimeTagger::setConditionalFilter (
    std::vector< channel_t > trigger,
    std::vector< channel_t > filtered,
    bool hardwareDelayCompensation = true ) [pure virtual]
```

configures the conditional filter

After each event on the trigger channels, one event per filtered channel will pass afterwards. This is handled in a very early stage in the pipeline, so all event limitations but the deadtime are suppressed. But the accuracy of the order of those events is low.

Refer the Manual for a description of this function.

#### Parameters

<i>trigger</i>	the channels that sets the condition
<i>filtered</i>	the channels that are filtered by the condition
<i>hardwareDelayCompensation</i>	if false, the physical hardware delay will not be compensated

#### 9.69.2.36 setEventDivider()

```
virtual void TimeTagger::setEventDivider (
    channel_t channel,
    unsigned int divider ) [pure virtual]
```

Divides the amount of transmitted edge per channel.

This filter decimates the events on a given channel by a specified. factor. So for a divider n, every nth event is transmitted through the filter and n-1 events are skipped between consecutive transmitted events. If a conditional filter is also active, the event divider is applied after the conditional filter, so the conditional is applied to the complete event stream and only events which pass the conditional filter are forwarded to the divider.

As it is a hardware filter, it reduces the required USB bandwidth and CPU processing power, but it cannot be configured for virtual channels.

## Parameters

<i>channel</i>	channel to be configured
<i>divider</i>	new divider, must be at least 1 and smaller than 65536

**9.69.2.37 setHardwareBufferSize()**

```
virtual void TimeTagger::setHardwareBufferSize (
    int size ) [pure virtual]
```

sets the maximum USB buffer size

This option controls the maximum buffer size of the USB connection. This can be used to balance low input latency vs high (peak) throughput.

## Parameters

<i>size</i>	the maximum buffer size in events
-------------	-----------------------------------

**9.69.2.38 setInputHysteresis()**

```
virtual void TimeTagger::setInputHysteresis (
    channel_t channel,
    int value ) [pure virtual]
```

configure the hysteresis voltage of the input comparator

Caution: This feature is only supported on the Time Tagger X The supported hysteresis voltages are 1 mV, 20 mV or 70 mV

## Parameters

<i>channel</i>	channel to be configured
<i>value</i>	the hysteresis voltage in milli Volt

**9.69.2.39 setInputImpedanceHigh()**

```
virtual void TimeTagger::setInputImpedanceHigh (
    channel_t channel,
    bool high_impedance ) [pure virtual]
```

enable high impedance termination mode

Caution: This feature is only supported on the Time Tagger X

## Parameters

<i>channel</i>	channel to be configured
<i>high_impedance</i>	set for the high impedance mode or cleared for the 50 Ohm termination mode

### 9.69.2.40 setInputMux()

```
virtual void TimeTagger::setInputMux (
    channel_t channel,
    int mux_mode ) [pure virtual]
```

configures the input multiplexer

Every physical input channel has an input multiplexer with 4 modes: 0: normal input mode 1: use the input from channel -1 (left) 2: use the input from channel +1 (right) 3: use the reference oscillator

Mode 1 and 2 cascades, so many inputs can be configured to get the same input events.

#### Parameters

<i>channel</i>	the physical channel of the input multiplexer
<i>mux_mode</i>	the configuration mode of the input multiplexer

### 9.69.2.41 setLED()

```
virtual void TimeTagger::setLED (
    uint32_t bitmask ) [pure virtual]
```

Enforce a state to the LEDs 0: led\_status[R] 16: led\_status[R] - mux 1: led\_status[G] 17: led\_status[G] - mux 2: led\_status[B] 18: led\_status[B] - mux 3: led\_power[R] 19: led\_power[R] - mux 4: led\_power[G] 20: led\_power[G] - mux 5: led\_power[B] 21: led\_power[B] - mux 6: led\_clock[R] 22: led\_clock[R] - mux 7: led\_clock[G] 23: led\_clock[G] - mux 8: led\_clock[B] 24: led\_clock[B] - mux.

### 9.69.2.42 setNormalization()

```
virtual void TimeTagger::setNormalization (
    std::vector< channel_t > channels,
    bool state ) [pure virtual]
```

enables or disables the normalization of the distribution.

Refer the Manual for a description of this function.

#### Parameters

<i>channels</i>	list of channels to modify
<i>state</i>	the new state

### 9.69.2.43 setSoundFrequency()

```
virtual void TimeTagger::setSoundFrequency (
    uint32_t freq_hz ) [pure virtual]
```

Set the Time Taggers internal buzzer to a frequency in Hz (freq\_hz==0 to disable)

## Parameters

<i>freq_hz</i>	the generated audio frequency
----------------	-------------------------------

**9.69.2.44 setTimeTaggerStreamBlockSize()**

```
virtual void TimeTagger::setStreamBlockSize (
    int max_events,
    int max_latency ) [pure virtual]
```

sets the maximum events and latency for the stream block size

This option controls the latency and the block size of the data stream. The default values are max\_events = 131072 events and max\_latency = 20 ms. Depending on which of the two parameters is exceeded first, the block stream size is adjusted accordingly. The block size will be reduced automatically for blocks when no signal is arriving for 512 ns on the Time Tagger Ultra and 1536 ns for the Time Tagger 20. \*

## Parameters

<i>max_events</i>	maximum number of events
<i>max_latency</i>	maximum latency in ms

**9.69.2.45 setTimeTaggerTestSignalDivider()**

```
virtual void TimeTagger::setTestSignalDivider (
    int divider ) [pure virtual]
```

set the divider for the frequency of the test signal

The base clock of the test signal oscillator for the Time Tagger Ultra is running at 100.8 MHz sampled down by an factor of 2 to have a similar base clock as the Time Tagger 20 (~50 MHz). The default divider is 63 -> ~800 kEvents/s. The base clock for the TTX is 333.3 MHz. The default divider is tuned to ~800 kEvents/s

## Parameters

<i>divider</i>	frequency divisor of the oscillator
----------------	-------------------------------------

**9.69.2.46 setTimeTaggerNetworkStreamCompression()**

```
virtual void TimeTagger::setTimeTaggerNetworkStreamCompression (
    bool active ) [pure virtual]
```

enable or disable additional compression of the timetag stream as sent over the network.

## Parameters

<i>active</i>	set if the compression is active or not.
---------------	--

**9.69.2.47 setTriggerLevel()**

```
virtual void TimeTagger::setTriggerLevel (
    channel_t channel,
    double voltage ) [pure virtual]
```

set the trigger voltage threshold of a channel

**Parameters**

<i>channel</i>	the channel to set
<i>voltage</i>	voltage level.. [0..1]

**9.69.2.48 startServer()**

```
virtual void TimeTagger::startServer (
    AccessMode access_mode,
    std::vector< channel_t > channels = std::vector< channel_t > (),
    uint32_t port = 41101 ) [pure virtual]
```

starts the Time Tagger server that will stream the time tags to the client.

**Parameters**

<i>access_mode</i>	set the type of access a user can have.
<i>port</i>	port at which this time tagger server will be listening on.
<i>channels</i>	channels to be streamed, if empty, all the channels will be exposed.

**9.69.2.49 stopServer()**

```
virtual void TimeTagger::stopServer ( ) [pure virtual]
```

stops the time tagger server if currently running, otherwise does nothing.

**9.69.2.50 updateBMCFirmware()**

```
virtual void TimeTagger::updateBMCFirmware (
    const std::string & firmware ) [pure virtual]
```

updates the firmware of the Time Tagger X board management controller

**Note**

The firmware is applied on the next power cycle of the device, *not* on pressing the power button.

**Parameters**

<i>firmware</i>	filename of the new firmware on disc
-----------------	--------------------------------------



### 9.69.2.51 xtra\_getAuxOut()

```
virtual bool TimeTagger::xtra_getAuxOut (
    int channel ) [pure virtual]
```

fetch the status of the aux out signal generator

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

#### Parameters

<i>channel</i>	select Aux Out 1 or 2
----------------	-----------------------

#### Returns

true if the aux out signal generator is enabled

### 9.69.2.52 xtra\_getAuxOutSignalDivider()

```
virtual int TimeTagger::xtra_getAuxOutSignalDivider (
    int channel ) [pure virtual]
```

get the divider for the frequency of the aux out signal generator

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

#### Parameters

<i>channel</i>	select Aux Out 1 or 2
----------------	-----------------------

#### Returns

the configured divider

### 9.69.2.53 xtra\_getAuxOutSignalDutyCycle()

```
virtual double TimeTagger::xtra_getAuxOutSignalDutyCycle (
    int channel ) [pure virtual]
```

get the dutycycle of the aux out signal generator

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

#### Parameters

<i>channel</i>	select Aux Out 1 or 2
----------------	-----------------------

**Returns**

the configured duty cycle

**9.69.2.54 xtra\_getAvgRisingFalling()**

```
virtual bool TimeTagger::xtra_getAvgRisingFalling (
    channel_t channel ) [pure virtual]
```

query if the rising and falling events shall be averaged

**Parameters**

<i>channel</i>	channel to be queried
----------------	-----------------------

**Returns**

if the rising and falling events shall be averaged

**9.69.2.55 xtra\_getClockAutoSelect()**

```
virtual bool TimeTagger::xtra_getClockAutoSelect ( ) [pure virtual]
```

queries if the auto clocking function is enabled

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

**Returns**

true if the external clock auto detection is enabled

**9.69.2.56 xtra\_getClockSource()**

```
virtual int TimeTagger::xtra_getClockSource ( ) [pure virtual]
```

fetch the overwritten reference clock source

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

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

**Returns**

selects the clock source

**9.69.2.57 xtra\_getHighPrioChannel()**

```
virtual bool TimeTagger::xtra_getHighPrioChannel (
    channel_t channel ) [pure virtual]
```

if this channel shall exit overflow regions

## Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

## Returns

if this channel shall exit overflow regions

**9.69.2.58 xtra\_measureTriggerLevel()**

```
virtual double TimeTagger::xtra_measureTriggerLevel (  
    channel_t channel ) [pure virtual]
```

measures the electrically applied the trigger voltage threshold of a channel

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

## Parameters

<i>channel</i>	the channel
----------------	-------------

## Returns

the voltage

**9.69.2.59 xtra\_setAuxOut()**

```
virtual void TimeTagger::xtra_setAuxOut (  
    int channel,  
    bool enabled ) [pure virtual]
```

enable or disable aux out

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

This will enable or disable the signal generator on the aux outputs.

## Parameters

<i>channel</i>	select Aux Out 1 or 2
<i>enabled</i>	enabled / disabled flag

**9.69.2.60 xtra\_setAuxOutSignal()**

```
virtual void TimeTagger::xtra_setAuxOutSignal (  
    int channel,
```

```
int divider,
double duty_cycle = 0.5 ) [pure virtual]
```

set the divider for the frequency of the aux out signal generator and enable aux out

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

The base clock for the TTX is 333.3 MHz.

#### Parameters

<i>channel</i>	select Aux Out 1 or 2
<i>divider</i>	frequency divisor of the oscillator
<i>duty_cycle</i>	the duty cycle ratio, will be clamped and rounded to an integer divisor

#### 9.69.2.61 xtra\_setAvgRisingFalling()

```
virtual void TimeTagger::xtra_setAvgRisingFalling (
    channel_t channel,
    bool enable ) [pure virtual]
```

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(channel)`, you can still fetch the original events there. However if both are configured to return the averaged result, the timestamps will be identical.

#### Parameters

<i>channel</i>	the channel, on which the average value shall be returned
<i>enable</i>	true if this channel shall yield the averaged timestamps

#### 9.69.2.62 xtra\_setClockAutoSelect()

```
virtual void TimeTagger::xtra_setClockAutoSelect (
    bool enabled ) [pure virtual]
```

activates auto clocking function

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

#### Parameters

<i>enabled</i>	true for auto detection of external clock
----------------	---

### 9.69.2.63 xtra\_setClockOut()

```
virtual void TimeTagger::xtra_setClockOut (
    bool enabled ) [pure virtual]
```

enables the clock output

Caution: this feature is for development purposes only and may not be part of future builds without further notice.

#### Parameters

<i>enabled</i>	true for enabling the 10 MHz clock output
----------------	---

### 9.69.2.64 xtra\_setClockSource()

```
virtual void TimeTagger::xtra_setClockSource (
    int source ) [pure virtual]
```

manually overwrite the reference clock source

0: internal clock 1: external 10 MHz 2: external 500 MHz

#### Parameters

<i>source</i>	selects the clock source
---------------	--------------------------

### 9.69.2.65 xtra\_setFanSpeed()

```
virtual void TimeTagger::xtra_setFanSpeed (
    double percentage = -1 ) [pure virtual]
```

configures the FAN speed on TTU HW  $\geq 1.3$

#### Parameters

<i>percentage</i>	the new speed, 0 means off, 100 means full on, negative means controlled.
-------------------	---

#### Note

This setting will get reset on USB errors.

### 9.69.2.66 xtra\_setHighPrioChannel()

```
virtual void TimeTagger::xtra_setHighPrioChannel (
    channel\_t channel,
    bool enable ) [pure virtual]
```

configures if this channel shall exit overflow regions.

If configured, each event of this channel within an overflow region will emit an OverflowEnd marker before the event and an OverflowBegin marker after the event. This can be used to split up regions by a slow trigger, e.g. for CountBetweenMarker usage.

#### Warning

Using this option disables the internal safety method for unrecoverable memory overflows. So only use this option on channels with a low datarate, else expect to get Error events, which invalidates the global time.

#### Parameters

<i>channel</i>	the channel, which shall be configured as high priority
<i>enable</i>	true if this channel shall have a high priority

The documentation for this class was generated from the following file:

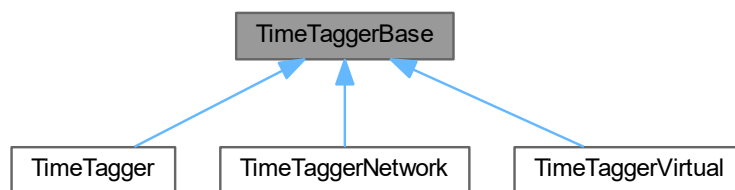
- [TimeTagger.h](#)

## 9.70 TimeTaggerBase Class Reference

Basis interface for all Time Tagger classes.

```
#include <TimeTagger.h>
```

Inheritance diagram for TimeTaggerBase:



#### Public Types

- typedef std::function< void([IteratorBase](#) \*) [IteratorCallback](#)>
- typedef std::map< [IteratorBase](#) \*, [IteratorCallback](#) > [IteratorCallbackMap](#)

## Public Member Functions

- virtual unsigned int [getFence](#) (bool alloc\_fence=true)=0  
*Generate a new fence object, which validates the current configuration and the current time.*
- virtual bool [waitForFence](#) (unsigned int fence, int64\_t timeout=-1)=0  
*Wait for a fence in the data stream.*
- virtual bool [sync](#) (int64\_t timeout=-1)=0  
*Sync the timetagger pipeline, so that all started iterators and their enabled channels are ready.*
- virtual [channel\\_t](#) [getInvertedChannel](#) ([channel\\_t](#) channel)=0  
*get the falling channel id for a rising channel and vice versa*
- virtual bool [isUnusedChannel](#) ([channel\\_t](#) channel)=0  
*compares the provided channel with CHANNEL\_UNUSED*
- virtual void [runSynchronized](#) (const [IteratorCallbackMap](#) &callbacks, bool block=true)=0  
*Run synchronized callbacks for a list of iterators.*
- virtual std::string [getConfiguration](#) ()=0  
*Fetches the overall configuration status of the Time Tagger object.*
- virtual void [setInputDelay](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void [setDelayHardware](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void [setDelaySoftware](#) ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual [timestamp\\_t](#) [getInputDelay](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [getDelaySoftware](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [getDelayHardware](#) ([channel\\_t](#) channel)=0  
*get time delay of a channel*
- virtual [timestamp\\_t](#) [setDeadtime](#) ([channel\\_t](#) channel, [timestamp\\_t](#) deadtime)=0  
*set the deadtime between two edges on the same channel.*
- virtual [timestamp\\_t](#) [getDeadtime](#) ([channel\\_t](#) channel)=0  
*get the deadtime between two edges on the same channel.*
- virtual void [setTestSignal](#) ([channel\\_t](#) channel, bool enabled)=0  
*enable/disable internal test signal on a channel.*
- virtual void [setTestSignal](#) (std::vector< [channel\\_t](#) > channel, bool enabled)=0  
*enable/disable internal test signal on multiple channels.*
- virtual bool [getTestSignal](#) ([channel\\_t](#) channel)=0  
*fetch the status of the test signal generator*
- virtual void [setSoftwareClock](#) ([channel\\_t](#) input\_channel, double input\_frequency=10e6, double averaging\_periods=1000, bool wait\_until\_locked=true)=0  
*enables a software PLL to lock the time to an external clock*
- virtual void [disableSoftwareClock](#) ()=0  
*disabled the software PLL*
- virtual [SoftwareClockState](#) [getSoftwareClockState](#) ()=0  
*queries all state information of the software clock*
- virtual long long [getOverflows](#) ()=0  
*get overflow count*
- virtual void [clearOverflows](#) ()=0  
*clear overflow counter*
- virtual long long [getOverflowsAndClear](#) ()=0  
*get and clear overflow counter*

## Protected Member Functions

- [TimeTaggerBase](#) ()  
*abstract interface class*
- virtual [~TimeTaggerBase](#) ()  
*destructor*
- [TimeTaggerBase](#) (const [TimeTaggerBase](#) &)=delete
- [TimeTaggerBase](#) & operator= (const [TimeTaggerBase](#) &)=delete
- virtual std::shared\_ptr< [IteratorBaseListNode](#) > [addIterator](#) ([IteratorBase](#) \*it)=0
- virtual void [freeIterator](#) ([IteratorBase](#) \*it)=0
- virtual [channel\\_t](#) [getNewVirtualChannel](#) ()=0
- virtual void [freeVirtualChannel](#) ([channel\\_t](#) channel)=0
- virtual void [registerChannel](#) ([channel\\_t](#) channel)=0  
*register a FPGA channel.*
- virtual void [registerChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [unregisterChannel](#) ([channel\\_t](#) channel)=0  
*release a previously registered channel.*
- virtual void [unregisterChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [addChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [removeChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [release](#) ()=0

### 9.70.1 Detailed Description

Basis interface for all Time Tagger classes.

This basis interface represents all common methods to add, remove, and run measurements.

### 9.70.2 Member Typedef Documentation

#### 9.70.2.1 IteratorCallback

```
typedef std::function<void(IteratorBase *)> TimeTaggerBase::IteratorCallback
```

#### 9.70.2.2 IteratorCallbackMap

```
typedef std::map<IteratorBase *, IteratorCallback> TimeTaggerBase::IteratorCallbackMap
```

### 9.70.3 Constructor & Destructor Documentation

#### 9.70.3.1 TimeTaggerBase() [1/2]

```
TimeTaggerBase::TimeTaggerBase ( ) [inline], [protected]
```

abstract interface class



### 9.70.3.2 ~TimeTaggerBase()

```
virtual TimeTaggerBase::~~TimeTaggerBase ( ) [inline], [protected], [virtual]
```

destructor

### 9.70.3.3 TimeTaggerBase() [2/2]

```
TimeTaggerBase::TimeTaggerBase (
    const TimeTaggerBase & ) [protected], [delete]
```

## 9.70.4 Member Function Documentation

### 9.70.4.1 addChild()

```
virtual void TimeTaggerBase::addChild (
    TimeTaggerBase * child ) [protected], [pure virtual]
```

### 9.70.4.2 addIterator()

```
virtual std::shared_ptr< IteratorBaseListNode > TimeTaggerBase::addIterator (
    IteratorBase * it ) [protected], [pure virtual]
```

### 9.70.4.3 clearOverflows()

```
virtual void TimeTaggerBase::clearOverflows ( ) [pure virtual]
```

clear overflow counter

Sets the overflow counter to zero

### 9.70.4.4 disableSoftwareClock()

```
virtual void TimeTaggerBase::disableSoftwareClock ( ) [pure virtual]
```

disabled the software PLL

See setSoftwareClock for further details.

### 9.70.4.5 freeIterator()

```
virtual void TimeTaggerBase::freeIterator (
    IteratorBase * it ) [protected], [pure virtual]
```

#### 9.70.4.6 freeVirtualChannel()

```
virtual void TimeTaggerBase::freeVirtualChannel (
    channel_t channel ) [protected], [pure virtual]
```

#### 9.70.4.7 getConfiguration()

```
virtual std::string TimeTaggerBase::getConfiguration ( ) [pure virtual]
```

Fetches the overall configuration status of the Time Tagger object.

##### Returns

a JSON serialized string with all configuration and status flags.

#### 9.70.4.8 getDeadtime()

```
virtual timestamp_t TimeTaggerBase::getDeadtime (
    channel_t channel ) [pure virtual]
```

get the deadtime between two edges on the same channel.

This function gets the user configurable deadtime.

##### Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

##### Returns

the real configured deadtime in picoseconds

#### 9.70.4.9 getDelayHardware()

```
virtual timestamp_t TimeTaggerBase::getDelayHardware (
    channel_t channel ) [pure virtual]
```

get time delay of a channel

see setDelayHardware

##### Parameters

<i>channel</i>	the channel
----------------	-------------

**Returns**

the hardware delay in picoseconds

**9.70.4.10 getDelaySoftware()**

```
virtual timestamp_t TimeTaggerBase::getDelaySoftware (
    channel_t channel ) [pure virtual]
```

get time delay of a channel

see setDelaySoftware

**Parameters**

<i>channel</i>	the channel
----------------	-------------

**Returns**

the software delay in picoseconds

**9.70.4.11 getFence()**

```
virtual unsigned int TimeTaggerBase::getFence (
    bool alloc_fence = true ) [pure virtual]
```

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 waitFence call were actually produced after the getFence call. The waitFence 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 amount of active fences.

**Parameters**

<i>alloc_fence</i>	if false, a reference to the most recently created fence will be returned instead
--------------------	---

**Returns**

the allocated fence

**9.70.4.12 getInputDelay()**

```
virtual timestamp_t TimeTaggerBase::getInputDelay (
    channel_t channel ) [pure virtual]
```

get time delay of a channel

see setInputDelay

**Parameters**

<i>channel</i>	the channel
----------------	-------------

**Returns**

the software delay in picoseconds

**9.70.4.13 getInvertedChannel()**

```
virtual channel_t TimeTaggerBase::getInvertedChannel (
    channel_t channel ) [pure virtual]
```

get the falling channel id for a rising channel and vice versa

If this channel has no inverted channel, UNUSED\_CHANNEL is returned. This is the case for most virtual channels.

**Parameters**

<i>channel</i>	The channel id to query
----------------	-------------------------

**Returns**

the inverted channel id

**9.70.4.14 getNewVirtualChannel()**

```
virtual channel_t TimeTaggerBase::getNewVirtualChannel ( ) [protected], [pure virtual]
```

**9.70.4.15 getOverflows()**

```
virtual long long TimeTaggerBase::getOverflows ( ) [pure virtual]
```

get overflow count

Get the number of communication overflows occurred

**9.70.4.16 getOverflowsAndClear()**

```
virtual long long TimeTaggerBase::getOverflowsAndClear ( ) [pure virtual]
```

get and clear overflow counter

Get the number of communication overflows occurred and sets them to zero

**9.70.4.17 getSoftwareClockState()**

```
virtual SoftwareClockState TimeTaggerBase::getSoftwareClockState ( ) [pure virtual]
```

queries all state information of the software clock

See setSoftwareClock for further details.

**9.70.4.18 getTestSignal()**

```
virtual bool TimeTaggerBase::getTestSignal (
    channel_t channel ) [pure virtual]
```

fetch the status of the test signal generator

**Parameters**

<i>channel</i>	the channel
----------------	-------------

Implemented in [TimeTaggerNetwork](#).

**9.70.4.19 isUnusedChannel()**

```
virtual bool TimeTaggerBase::isUnusedChannel (
    channel_t channel ) [pure virtual]
```

compares the provided channel with CHANNEL\_UNUSED

But also keeps care about the channel number scheme and selects either CHANNEL\_UNUSED or CHANNEL\_UNUSED\_OLD

**9.70.4.20 operator=()**

```
TimeTaggerBase & TimeTaggerBase::operator= (
    const TimeTaggerBase & ) [protected], [delete]
```

**9.70.4.21 registerChannel() [1/2]**

```
virtual void TimeTaggerBase::registerChannel (
    channel_t channel ) [protected], [pure virtual]
```

register a FPGA channel.

Only events on previously registered channels will be transferred over the communication channel.

**Parameters**

<i>channel</i>	the channel
----------------	-------------

**9.70.4.22 registerChannel()** [2/2]

```
virtual void TimeTaggerBase::registerChannel (
    std::set< channel_t > channels ) [protected], [pure virtual]
```

**9.70.4.23 release()**

```
virtual void TimeTaggerBase::release ( ) [protected], [pure virtual]
```

**9.70.4.24 removeChild()**

```
virtual void TimeTaggerBase::removeChild (
    TimeTaggerBase * child ) [protected], [pure virtual]
```

**9.70.4.25 runSynchronized()**

```
virtual void TimeTaggerBase::runSynchronized (
    const IteratorCallbackMap & callbacks,
    bool block = true ) [pure virtual]
```

Run synchronized callbacks for a list of iterators.

This method has a list of callbacks for a list of iterators. Those callbacks are called for a synchronized data set, but in parallel. They are called from an internal worker thread. As the data set is synchronized, this creates a bottleneck for one worker thread, so only fast and non-blocking callbacks are allowed.

**Parameters**

<i>callbacks</i>	Map of callbacks per iterator
<i>block</i>	Shall this method block until all callbacks are finished

**9.70.4.26 setDeadtime()**

```
virtual timestamp_t TimeTaggerBase::setDeadtime (
    channel_t channel,
    timestamp_t deadtime ) [pure virtual]
```

set the deadtime between two edges on the same channel.

This function sets the user configurable deadtime. The requested time will be rounded to the nearest multiple of the clock time. The deadtime will also be clamped to device specific limitations.

As the actual deadtime will be altered, the real value will be returned.

**Parameters**

<i>channel</i>	channel to be configured
<i>deadtime</i>	new deadtime in picoseconds

**Returns**

the real configured deadtime in picoseconds

**9.70.4.27 setDelayHardware()**

```
virtual void TimeTaggerBase::setDelayHardware (
    channel_t channel,
    timestamp_t delay ) [pure virtual]
```

set time delay on a channel

When set, every event on this physical input channel is delayed by the given delay in picoseconds. This delay is implemented on the hardware before any filter with no performance overhead. The maximum delay on the Time Tagger Ultra series is 2 us. This affects both the rising and the falling event at the same time.

**Parameters**

<i>channel</i>	the channel to set
<i>delay</i>	the hardware delay in picoseconds

**9.70.4.28 setDelaySoftware()**

```
virtual void TimeTaggerBase::setDelaySoftware (
    channel_t channel,
    timestamp_t delay ) [pure virtual]
```

set time delay on a channel

When set, every event on this channel is delayed by the given delay in picoseconds. This happens on the computer and so after the on-device filters. Please use `setDelayHardware` instead for better performance. This affects either the the rising or the falling event only.

This method has the best performance with "small delays". The delay is considered "small" when less than 100 events arrive within the time of the largest delay set. For example, if the total event-rate over all channels used is 10 Mevent/s, the signal can be delayed efficiently up to 10 microseconds. For large delays, please use [DelayedChannel](#) instead.

**Parameters**

<i>channel</i>	the channel to set
<i>delay</i>	the software delay in picoseconds

**9.70.4.29 setInputDelay()**

```
virtual void TimeTaggerBase::setInputDelay (
    channel_t channel,
    timestamp_t delay ) [pure virtual]
```

set time delay on a channel

When set, every event on this channel is delayed by the given delay in picoseconds.

This method has the best performance with "small delays". The delay is considered "small" when less than 100 events arrive within the time of the largest delay set. For example, if the total event-rate over all channels used is 10 Mevent/s, the signal can be delayed efficiently up to 10 microseconds. For large delays, please use [DelayedChannel](#) instead.

#### Parameters

<i>channel</i>	the channel to set
<i>delay</i>	the delay in picoseconds

#### 9.70.4.30 setSoftwareClock()

```
virtual void TimeTaggerBase::setSoftwareClock (
    channel_t input_channel,
    double input_frequency = 10e6,
    double averaging_periods = 1000,
    bool wait_until_locked = true ) [pure virtual]
```

enables a software PLL to lock the time to an external clock

This feature implements a software PLL on the CPU. This can replace external clocks with no restrictions on correlated data to other inputs. It uses a first-order loop filter to ignore the discretization noise of the input and to provide some kind of cutoff frequency when to apply the extern clock.

#### Note

Within the first  $100 * \text{averaging\_factor} * \text{clock\_period}$ , a frequency locking approach is applied. The phase gets locked afterwards.

#### Parameters

<i>input_channel</i>	The physical input channel
<i>input_frequency</i>	Frequency of the configured external clock. Slight variations will be canceled out. Defaults to 10e6 for 10 MHz
<i>averaging_periods</i>	Times clock_period is the cutoff period for the filter. Shorter periods are evaluated with the Time Tagger's internal clock, longer periods are evaluated with the here configured external clock
<i>wait_until_locked</i>	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.

#### 9.70.4.31 setTestSignal() [1/2]

```
virtual void TimeTaggerBase::setTestSignal (
    channel_t channel,
    bool enabled ) [pure virtual]
```

enable/disable internal test signal on a channel.

This will connect or disconnect the channel with the on-chip uncorrelated signal generator.



## Parameters

<i>channel</i>	the channel
<i>enabled</i>	enabled / disabled flag

**9.70.4.32 setTestSignal()** [2/2]

```
virtual void TimeTaggerBase::setTestSignal (
    std::vector< channel_t > channel,
    bool enabled ) [pure virtual]
```

enable/disable internal test signal on multiple channels.

This will connect or disconnect the channels with the on-chip uncorrelated signal generator.

## Parameters

<i>channel</i>	a vector of channels
<i>enabled</i>	enabled / disabled flag

**9.70.4.33 sync()**

```
virtual bool TimeTaggerBase::sync (
    int64_t timeout = -1 ) [pure virtual]
```

Sync the timetagger pipeline, so that all started iterators and their enabled channels are ready.

This is a shortcut for calling getFence and waitForFence at once. See getFence for more details.

## Parameters

<i>timeout</i>	timeout in milliseconds. Negative means no timeout, zero returns immediately.
----------------	---

## Returns

true on success, false on timeout

**9.70.4.34 unregisterChannel()** [1/2]

```
virtual void TimeTaggerBase::unregisterChannel (
    channel_t channel ) [protected], [pure virtual]
```

release a previously registered channel.

## Parameters

<i>channel</i>	the channel
----------------	-------------

### 9.70.4.35 unregisterChannel() [2/2]

```
virtual void TimeTaggerBase::unregisterChannel (
    std::set< channel_t > channels ) [protected], [pure virtual]
```

### 9.70.4.36 waitForFence()

```
virtual bool TimeTaggerBase::waitForFence (
    unsigned int fence,
    int64_t timeout = -1 ) [pure virtual]
```

Wait for a fence in the data stream.

See `getFence` for more details.

#### Parameters

<i>fence</i>	fence object, which shall be waited on
<i>timeout</i>	timeout in milliseconds. Negative means no timeout, zero returns immediately.

#### Returns

true if the fence has passed, false on timeout

The documentation for this class was generated from the following file:

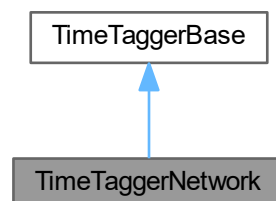
- [TimeTagger.h](#)

## 9.71 TimeTaggerNetwork Class Reference

network [TimeTagger](#) client.

```
#include <TimeTagger.h>
```

Inheritance diagram for TimeTaggerNetwork:



## Public Member Functions

- virtual bool `isConnected` ()=0  
*check if the network time tagger is currently connected to a server*
- virtual void `setTriggerLevel` (`channel_t` channel, double voltage)=0  
*set the trigger voltage threshold of a channel*
- virtual double `getTriggerLevel` (`channel_t` channel)=0  
*get the trigger voltage threshold of a channel*
- virtual void `setConditionalFilter` (std::vector< `channel_t` > trigger, std::vector< `channel_t` > filtered, bool hardwareDelayCompensation=true)=0  
*configures the conditional filter*
- virtual void `clearConditionalFilter` ()=0  
*deactivates the conditional filter*
- virtual std::vector< `channel_t` > `getConditionalFilterTrigger` ()=0  
*fetches the configuration of the conditional filter*
- virtual std::vector< `channel_t` > `getConditionalFilterFiltered` ()=0  
*fetches the configuration of the conditional filter*
- virtual void `setTestSignalDivider` (int divider)=0  
*set the divider for the frequency of the test signal*
- virtual int `getTestSignalDivider` ()=0  
*get the divider for the frequency of the test signal*
- virtual bool `getTestSignal` (`channel_t` channel)=0  
*fetch the status of the test signal generator*
- virtual void `setDelayClient` (`channel_t` channel, `timestamp_t` time)=0  
*set time delay on a channel*
- virtual `timestamp_t` `getDelayClient` (`channel_t` channel)=0  
*get the time delay of a channel*
- virtual `timestamp_t` `getHardwareDelayCompensation` (`channel_t` channel)=0  
*get hardware delay compensation of a channel*
- virtual void `setNormalization` (std::vector< `channel_t` > channels, bool state)=0  
*enables or disables the normalization of the distribution.*
- virtual bool `getNormalization` (`channel_t` channel)=0  
*returns the the normalization of the distribution.*
- virtual void `setHardwareBufferSize` (int size)=0  
*sets the maximum USB buffer size*
- virtual int `getHardwareBufferSize` ()=0  
*queries the size of the USB queue*
- virtual void `setStreamBlockSize` (int max\_events, int max\_latency)=0  
*sets the maximum events and latency for the stream block size*
- virtual int `getStreamBlockSizeEvents` ()=0
- virtual int `getStreamBlockSizeLatency` ()=0
- virtual void `setEventDivider` (`channel_t` channel, unsigned int divider)=0  
*Divides the amount of transmitted edge per channel.*
- virtual unsigned int `getEventDivider` (`channel_t` channel)=0  
*Returns the factor of the dividing filter.*
- virtual std::string `getSerial` ()=0  
*identifies the hardware by serial number*
- virtual std::string `getModel` ()=0  
*identifies the hardware by Time Tagger Model*
- virtual int `getChannelNumberScheme` ()=0  
*Fetch the configured numbering scheme for this `TimeTagger` object.*

- virtual std::vector< double > [getDACRange](#) ()=0  
*returns the minimum and the maximum voltage of the DACs as a trigger reference*
- virtual std::vector< [channel\\_t](#) > [getChannelList](#) ([ChannelEdge](#) type=[ChannelEdge::All](#))=0  
*fetch a vector of all physical input channel ids*
- virtual [timestamp\\_t](#) [getPsPerClock](#) ()=0  
*fetch the duration of each clock cycle in picoseconds*
- virtual std::string [getPcbVersion](#) ()=0  
*Return the hardware version of the PCB board. Version 0 is everything before mid 2018 and with the channel configuration ZERO. version >= 1 is channel configuration ONE.*
- virtual std::string [getFirmwareVersion](#) ()=0  
*Return an unique identifier for the applied firmware.*
- virtual std::string [getSensorData](#) ()=0  
*Show the status of the sensor data from the FPGA and peripherals on the console.*
- virtual void [setLED](#) (uint32\_t bitmask)=0  
*Enforce a state to the LEDs 0: led\_status[R] 16: led\_status[R] - mux 1: led\_status[G] 17: led\_status[G] - mux 2: led\_status[B] 18: led\_status[B] - mux 3: led\_power[R] 19: led\_power[R] - mux 4: led\_power[G] 20: led\_power[G] - mux 5: led\_power[B] 21: led\_power[B] - mux 6: led\_clock[R] 22: led\_clock[R] - mux 7: led\_clock[G] 23: led\_clock[G] - mux 8: led\_clock[B] 24: led\_clock[B] - mux.*
- virtual std::string [getDeviceLicense](#) ()=0  
*gets the license, installed on this device currently*
- virtual void [setSoundFrequency](#) (uint32\_t freq\_hz)=0  
*Set the Time Taggers internal buzzer to a frequency in Hz (freq\_hz==0 to disable)*
- virtual void [setTimeTaggerNetworkStreamCompression](#) (bool active)=0  
*enable or disable additional compression of the timetag stream as sent over the network.*
- virtual long long [getOverflowsClient](#) ()=0
- virtual void [clearOverflowsClient](#) ()=0
- virtual long long [getOverflowsAndClearClient](#) ()=0
- virtual void [setInputImpedanceHigh](#) ([channel\\_t](#) channel, bool high\_impedance)=0  
*enable high impedance termination mode*
- virtual bool [getInputImpedanceHigh](#) ([channel\\_t](#) channel)=0  
*query the state of the high impedance termination mode*
- virtual void [setInputHysteresis](#) ([channel\\_t](#) channel, int value)=0  
*configure the hysteresis voltage of the input comparator*
- virtual int [getInputHysteresis](#) ([channel\\_t](#) channel)=0  
*query the hysteresis voltage of the input comparator*

## Public Member Functions inherited from [TimeTaggerBase](#)

- virtual unsigned int [getFence](#) (bool alloc\_fence=true)=0  
*Generate a new fence object, which validates the current configuration and the current time.*
- virtual bool [waitForFence](#) (unsigned int fence, int64\_t timeout=-1)=0  
*Wait for a fence in the data stream.*
- virtual bool [sync](#) (int64\_t timeout=-1)=0  
*Sync the timetagger pipeline, so that all started iterators and their enabled channels are ready.*
- virtual [channel\\_t](#) [getInvertedChannel](#) ([channel\\_t](#) channel)=0  
*get the falling channel id for a rising channel and vice versa*
- virtual bool [isUnusedChannel](#) ([channel\\_t](#) channel)=0  
*compares the provided channel with CHANNEL\_UNUSED*
- virtual void [runSynchronized](#) (const [IteratorCallbackMap](#) &callbacks, bool block=true)=0  
*Run synchronized callbacks for a list of iterators.*
- virtual std::string [getConfiguration](#) ()=0

- Fetches the overall configuration status of the Time Tagger object.*
- virtual void `setInputDelay` (`channel_t` channel, `timestamp_t` delay)=0  
*set time delay on a channel*
- virtual void `setDelayHardware` (`channel_t` channel, `timestamp_t` delay)=0  
*set time delay on a channel*
- virtual void `setDelaySoftware` (`channel_t` channel, `timestamp_t` delay)=0  
*set time delay on a channel*
- virtual `timestamp_t` `getInputDelay` (`channel_t` channel)=0  
*get time delay of a channel*
- virtual `timestamp_t` `getDelaySoftware` (`channel_t` channel)=0  
*get time delay of a channel*
- virtual `timestamp_t` `getDelayHardware` (`channel_t` channel)=0  
*get time delay of a channel*
- virtual `timestamp_t` `setDeadtime` (`channel_t` channel, `timestamp_t` deadtime)=0  
*set the deadtime between two edges on the same channel.*
- virtual `timestamp_t` `getDeadtime` (`channel_t` channel)=0  
*get the deadtime between two edges on the same channel.*
- virtual void `setTestSignal` (`channel_t` channel, bool enabled)=0  
*enable/disable internal test signal on a channel.*
- virtual void `setTestSignal` (std::vector< `channel_t` > channel, bool enabled)=0  
*enable/disable internal test signal on multiple channels.*
- virtual void `setSoftwareClock` (`channel_t` input\_channel, double input\_frequency=10e6, double averaging\_↔ periods=1000, bool wait\_until\_locked=true)=0  
*enables a software PLL to lock the time to an external clock*
- virtual void `disableSoftwareClock` ()=0  
*disabled the software PLL*
- virtual `SoftwareClockState` `getSoftwareClockState` ()=0  
*queries all state information of the software clock*
- virtual long long `getOverflows` ()=0  
*get overflow count*
- virtual void `clearOverflows` ()=0  
*clear overflow counter*
- virtual long long `getOverflowsAndClear` ()=0  
*get and clear overflow counter*

### Additional Inherited Members

### Public Types inherited from `TimeTaggerBase`

- typedef std::function< void(`IteratorBase` \*) `IteratorCallback`>
- typedef std::map< `IteratorBase` \*, `IteratorCallback` > `IteratorCallbackMap`

## Protected Member Functions inherited from [TimeTaggerBase](#)

- [TimeTaggerBase](#) ()  
*abstract interface class*
- virtual [~TimeTaggerBase](#) ()  
*destructor*
- [TimeTaggerBase](#) (const [TimeTaggerBase](#) &)=delete
- [TimeTaggerBase](#) & operator= (const [TimeTaggerBase](#) &)=delete
- virtual std::shared\_ptr< [IteratorBaseListNode](#) > [addIterator](#) ([IteratorBase](#) \*it)=0
- virtual void [freeIterator](#) ([IteratorBase](#) \*it)=0
- virtual [channel\\_t](#) [getNewVirtualChannel](#) ()=0
- virtual void [freeVirtualChannel](#) ([channel\\_t](#) channel)=0
- virtual void [registerChannel](#) ([channel\\_t](#) channel)=0  
*register a FPGA channel.*
- virtual void [registerChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [unregisterChannel](#) ([channel\\_t](#) channel)=0  
*release a previously registered channel.*
- virtual void [unregisterChannel](#) (std::set< [channel\\_t](#) > channels)=0
- virtual void [addChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [removeChild](#) ([TimeTaggerBase](#) \*child)=0
- virtual void [release](#) ()=0

### 9.71.1 Detailed Description

network [TimeTagger](#) client.

The [TimeTaggerNetwork](#) class is a client that implements access to the Time Tagger server. [TimeTaggerNetwork](#) receives the time-tag stream from the Time Tagger server over the network and provides an interface for controlling connection and the Time Tagger hardware. Instance of this class can be transparently used to create measurements, virtual channels and other [Iterator](#) instances.

### 9.71.2 Member Function Documentation

#### 9.71.2.1 [clearConditionalFilter\(\)](#)

```
virtual void TimeTaggerNetwork::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to [setConditionalFilter](#)({},{})

#### 9.71.2.2 [clearOverflowsClient\(\)](#)

```
virtual void TimeTaggerNetwork::clearOverflowsClient ( ) [pure virtual]
```

### 9.71.2.3 getChannelList()

```
virtual std::vector< channel_t > TimeTaggerNetwork::getChannelList (
    ChannelEdge type = ChannelEdge::All ) [pure virtual]
```

fetch a vector of all physical input channel ids

The function returns the channel of all rising and falling edges. For example for the Time Tagger 20 (8 input channels) TT\_CHANNEL\_NUMBER\_SCHEME\_ZERO: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15} and for TT\_CHANNEL\_NUMBER\_SCHEME\_ONE: {-8,-7,-6,-5,-4,-3,-2,-1,1,2,3,4,5,6,7,8}

TT\_CHANNEL\_RISING\_EDGES returns only the rising edges SCHEME\_ONE: {1,2,3,4,5,6,7,8} and TT\_CHANNEL\_FALLING\_EDGES return only the falling edges SCHEME\_ONE: {-1,-2,-3,-4,-5,-6,-7,-8} which are the invertedChannels of the rising edges.

### 9.71.2.4 getChannelNumberScheme()

```
virtual int TimeTaggerNetwork::getChannelNumberScheme ( ) [pure virtual]
```

Fetch the configured numbering scheme for this [TimeTagger](#) object.

Please see [setTimeTaggerChannelNumberScheme\(\)](#) for details.

### 9.71.2.5 getConditionalFilterFiltered()

```
virtual std::vector< channel_t > TimeTaggerNetwork::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

### 9.71.2.6 getConditionalFilterTrigger()

```
virtual std::vector< channel_t > TimeTaggerNetwork::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

### 9.71.2.7 getDACRange()

```
virtual std::vector< double > TimeTaggerNetwork::getDACRange ( ) [pure virtual]
```

returns the minimum and the maximum voltage of the DACs as a trigger reference

### 9.71.2.8 getDelayClient()

```
virtual timestamp_t TimeTaggerNetwork::getDelayClient (
    channel_t channel ) [pure virtual]
```

get the time delay of a channel

see setDelayClient

**Parameters**

<i>channel</i>	the channel
----------------	-------------

**Returns**

the software delay in picoseconds

**9.71.2.9 getDeviceLicense()**

```
virtual std::string TimeTaggerNetwork::getDeviceLicense ( ) [pure virtual]
```

gets the license, installed on this device currently

**Returns**

a JSON string containing the current device license

**9.71.2.10 getEventDivider()**

```
virtual unsigned int TimeTaggerNetwork::getEventDivider (
    channel_t channel ) [pure virtual]
```

Returns the factor of the dividing filter.

See setEventDivider for further details.

**Parameters**

<i>channel</i>	channel to be queried
----------------	-----------------------

**Returns**

the configured divider

**9.71.2.11 getFirmwareVersion()**

```
virtual std::string TimeTaggerNetwork::getFirmwareVersion ( ) [pure virtual]
```

Return an unique identifier for the applied firmware.

This function returns a comma separated list of the firmware version with

- the device identifier: TT-20 or TT-Ultra
- the firmware identifier: FW 3
- optional the timestamp of the assembling of the firmware
- the firmware identifier of the USB chip: OK 1.30 eg "TT-Ultra, FW 3, TS 2018-11-13 22:57:32, OK 1.30"



#### 9.71.2.12 getHardwareBufferSize()

```
virtual int TimeTaggerNetwork::getHardwareBufferSize ( ) [pure virtual]
```

queries the size of the USB queue

See setHardwareBufferSize for more information.

##### Returns

the actual size of the USB queue in events

#### 9.71.2.13 getHardwareDelayCompensation()

```
virtual timestamp_t TimeTaggerNetwork::getHardwareDelayCompensation (
    channel_t channel ) [pure virtual]
```

get 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

<i>channel</i>	the channel
----------------	-------------

##### Returns

the hardware delay compensation in picoseconds

#### 9.71.2.14 getInputHysteresis()

```
virtual int TimeTaggerNetwork::getInputHysteresis (
    channel_t channel ) [pure virtual]
```

query the hysteresis voltage of the input comparator

##### Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

##### Returns

the hysteresis voltage in milli Volt

#### 9.71.2.15 getInputImpedanceHigh()

```
virtual bool TimeTaggerNetwork::getInputImpedanceHigh (
```

```
channel_t channel ) [pure virtual]
```

query the state of the high impedance termination mode

#### Parameters

<i>channel</i>	channel to be queried
----------------	-----------------------

#### Returns

true for the high impedance mode or false for the 50 Ohm termination mode

#### 9.71.2.16 getModel()

```
virtual std::string TimeTaggerNetwork::getModel ( ) [pure virtual]
```

identifies the hardware by Time Tagger Model

#### 9.71.2.17 getNormalization()

```
virtual bool TimeTaggerNetwork::getNormalization (
    channel_t channel ) [pure virtual]
```

returns the the normalization of the distribution.

Refer the Manual for a description of this function.

#### Parameters

<i>channel</i>	the channel to query
----------------	----------------------

#### Returns

if the normalization is enabled

#### 9.71.2.18 getOverflowsAndClearClient()

```
virtual long long TimeTaggerNetwork::getOverflowsAndClearClient ( ) [pure virtual]
```

#### 9.71.2.19 getOverflowsClient()

```
virtual long long TimeTaggerNetwork::getOverflowsClient ( ) [pure virtual]
```

#### 9.71.2.20 getPcbVersion()

```
virtual std::string TimeTaggerNetwork::getPcbVersion ( ) [pure virtual]
```

Return the hardware version of the PCB board. Version 0 is everything before mid 2018 and with the channel configuration ZERO. version  $\geq 1$  is channel configuration ONE.

#### 9.71.2.21 getPsPerClock()

```
virtual timestamp_t TimeTaggerNetwork::getPsPerClock ( ) [pure virtual]
```

fetch the duration of each clock cycle in picoseconds

#### 9.71.2.22 getSensorData()

```
virtual std::string TimeTaggerNetwork::getSensorData ( ) [pure virtual]
```

Show the status of the sensor data from the FPGA and peripherals on the console.

#### 9.71.2.23 getSerial()

```
virtual std::string TimeTaggerNetwork::getSerial ( ) [pure virtual]
```

identifies the hardware by serial number

#### 9.71.2.24 getStreamBlockSizeEvents()

```
virtual int TimeTaggerNetwork::getStreamBlockSizeEvents ( ) [pure virtual]
```

#### 9.71.2.25 getStreamBlockSizeLatency()

```
virtual int TimeTaggerNetwork::getStreamBlockSizeLatency ( ) [pure virtual]
```

#### 9.71.2.26 getTestSignal()

```
virtual bool TimeTaggerNetwork::getTestSignal (
    channel_t channel ) [pure virtual]
```

fetch the status of the test signal generator

##### Parameters

<i>channel</i>	the channel
----------------	-------------

Implements [TimeTaggerBase](#).

#### 9.71.2.27 getTestSignalDivider()

```
virtual int TimeTaggerNetwork::getTestSignalDivider ( ) [pure virtual]
```

get the divider for the frequency of the test signal

### 9.71.2.28 getTriggerLevel()

```
virtual double TimeTaggerNetwork::getTriggerLevel (
    channel_t channel ) [pure virtual]
```

get the trigger voltage threshold of a channel

#### Parameters

<i>channel</i>	the channel
----------------	-------------

### 9.71.2.29 isConnected()

```
virtual bool TimeTaggerNetwork::isConnected ( ) [pure virtual]
```

check if the network time tagger is currently connected to a server

#### Returns

returns true if it's currently connected to a server; false, otherwise.

### 9.71.2.30 setConditionalFilter()

```
virtual void TimeTaggerNetwork::setConditionalFilter (
    std::vector< channel_t > trigger,
    std::vector< channel_t > filtered,
    bool hardwareDelayCompensation = true ) [pure virtual]
```

configures the conditional filter

After each event on the trigger channels, one event per filtered channel will pass afterwards. This is handled in a very early stage in the pipeline, so all event limitations but the deadtime are suppressed. But the accuracy of the order of those events is low.

Refer the Manual for a description of this function.

#### Parameters

<i>trigger</i>	the channels that sets the condition
<i>filtered</i>	the channels that are filtered by the condition
<i>hardwareDelayCompensation</i>	if false, the physical hardware delay will not be compensated

### 9.71.2.31 setDelayClient()

```
virtual void TimeTaggerNetwork::setDelayClient (
    channel_t channel,
    timestamp_t time ) [pure virtual]
```

set time delay on a channel

When set, every event on this channel is delayed by the given delay in picoseconds.

This delay is implemented on the client and does not affect the server nor requires the Control flag.

#### Parameters

<i>channel</i>	the channel to set
<i>time</i>	the delay in picoseconds

### 9.71.2.32 setEventDivider()

```
virtual void TimeTaggerNetwork::setEventDivider (
    channel_t channel,
    unsigned int divider ) [pure virtual]
```

Divides the amount of transmitted edge per channel.

This filter decimates the events on a given channel by a specified factor. So for a divider  $n$ , every  $n$ th event is transmitted through the filter and  $n-1$  events are skipped between consecutive transmitted events. If a conditional filter is also active, the event divider is applied after the conditional filter, so the conditional is applied to the complete event stream and only events which pass the conditional filter are forwarded to the divider.

As it is a hardware filter, it reduces the required USB bandwidth and CPU processing power, but it cannot be configured for virtual channels.

#### Parameters

<i>channel</i>	channel to be configured
<i>divider</i>	new divider, must be at least 1 and smaller than 65536

### 9.71.2.33 setHardwareBufferSize()

```
virtual void TimeTaggerNetwork::setHardwareBufferSize (
    int size ) [pure virtual]
```

sets the maximum USB buffer size

This option controls the maximum buffer size of the USB connection. This can be used to balance low input latency vs high (peak) throughput.

#### Parameters

<i>size</i>	the maximum buffer size in events
-------------	-----------------------------------

### 9.71.2.34 setInputHysteresis()

```
virtual void TimeTaggerNetwork::setInputHysteresis (
```

```
channel_t channel,
int value ) [pure virtual]
```

configure the hysteresis voltage of the input comparator

Caution: This feature is only supported on the Time Tagger X The supported hysteresis voltages are 1 mV, 20 mV or 70 mV

#### Parameters

<i>channel</i>	channel to be configured
<i>value</i>	the hysteresis voltage in milli Volt

#### 9.71.2.35 setInputImpedanceHigh()

```
virtual void TimeTaggerNetwork::setInputImpedanceHigh (
channel_t channel,
bool high_impedance ) [pure virtual]
```

enable high impedance termination mode

Caution: This feature is only supported on the Time Tagger X

#### Parameters

<i>channel</i>	channel to be configured
<i>high_impedance</i>	set for the high impedance mode or cleared for the 50 Ohm termination mode

#### 9.71.2.36 setLED()

```
virtual void TimeTaggerNetwork::setLED (
uint32_t bitmask ) [pure virtual]
```

Enforce a state to the LEDs 0: led\_status[R] 16: led\_status[R] - mux 1: led\_status[G] 17: led\_status[G] - mux 2: led\_status[B] 18: led\_status[B] - mux 3: led\_power[R] 19: led\_power[R] - mux 4: led\_power[G] 20: led\_power[G] - mux 5: led\_power[B] 21: led\_power[B] - mux 6: led\_clock[R] 22: led\_clock[R] - mux 7: led\_clock[G] 23: led\_clock[G] - mux 8: led\_clock[B] 24: led\_clock[B] - mux.

#### 9.71.2.37 setNormalization()

```
virtual void TimeTaggerNetwork::setNormalization (
std::vector< channel_t > channels,
bool state ) [pure virtual]
```

enables or disables the normalization of the distribution.

Refer the Manual for a description of this function.

## Parameters

<i>channels</i>	list of channels to modify
<i>state</i>	the new state

**9.71.2.38 setSoundFrequency()**

```
virtual void TimeTaggerNetwork::setSoundFrequency (
    uint32_t freq_hz ) [pure virtual]
```

Set the Time Taggers internal buzzer to a frequency in Hz (freq\_hz==0 to disable)

## Parameters

<i>freq_hz</i>	the generated audio frequency
----------------	-------------------------------

**9.71.2.39 setStreamBlockSize()**

```
virtual void TimeTaggerNetwork::setStreamBlockSize (
    int max_events,
    int max_latency ) [pure virtual]
```

sets the maximum events and latency for the stream block size

This option controls the latency and the block size of the data stream. The default values are max\_events = 131072 events and max\_latency = 20 ms. Depending on which of the two parameters is exceeded first, the block stream size is adjusted accordingly. The block size will be reduced automatically for blocks when no signal is arriving for 512 ns on the Time Tagger Ultra and 1536 ns for the Time Tagger 20. \*

## Parameters

<i>max_events</i>	maximum number of events
<i>max_latency</i>	maximum latency in ms

**9.71.2.40 setTestSignalDivider()**

```
virtual void TimeTaggerNetwork::setTestSignalDivider (
    int divider ) [pure virtual]
```

set the divider for the frequency of the test signal

The base clock of the test signal oscillator for the Time Tagger Ultra is running at 100.8 MHz sampled down by an factor of 2 to have a similar base clock as the Time Tagger 20 (~50 MHz). The default divider is 63 -> ~800 kEvents/s

## Parameters

<i>divider</i>	frequency divisor of the oscillator
----------------	-------------------------------------

### 9.71.2.41 setTimeTaggerNetworkStreamCompression()

```
virtual void TimeTaggerNetwork::setTimeTaggerNetworkStreamCompression (
    bool active ) [pure virtual]
```

enable or disable additional compression of the timetag stream as sent over the network.

#### Parameters

<i>active</i>	set if the compression is active or not.
---------------	--

### 9.71.2.42 setTriggerLevel()

```
virtual void TimeTaggerNetwork::setTriggerLevel (
    channel_t channel,
    double voltage ) [pure virtual]
```

set the trigger voltage threshold of a channel

#### Parameters

<i>channel</i>	the channel to set
<i>voltage</i>	voltage level.. [0..1]

The documentation for this class was generated from the following file:

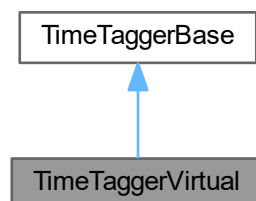
- [TimeTagger.h](#)

## 9.72 TimeTaggerVirtual Class Reference

virtual [TimeTagger](#) based on dump files

```
#include <TimeTagger.h>
```

Inheritance diagram for TimeTaggerVirtual:





## Public Member Functions

- virtual uint64\_t **replay** (const std::string &file, [timestamp\\_t](#) begin=0, [timestamp\\_t](#) duration=-1, bool queue=true)=0  
*replay a given dump file on the disc*
- virtual void **stop** ()=0  
*stops the current and all queued files.*
- virtual void **reset** ()=0  
*stops the all queued files and resets the [TimeTaggerVirtual](#) to its default settings*
- virtual bool **waitForCompletion** (uint64\_t ID=0, int64\_t timeout=-1)=0  
*block the current thread until the replay finish*
- virtual void **setReplaySpeed** (double speed)=0  
*configures the speed factor for the virtual tagger.*
- virtual double **getReplaySpeed** ()=0  
*fetches the speed factor*
- virtual void **setConditionalFilter** (std::vector< [channel\\_t](#) > trigger, std::vector< [channel\\_t](#) > filtered)=0  
*configures the conditional filter*
- virtual void **clearConditionalFilter** ()=0  
*deactivates the conditional filter*
- virtual std::vector< [channel\\_t](#) > **getConditionalFilterTrigger** ()=0  
*fetches the configuration of the conditional filter*
- virtual std::vector< [channel\\_t](#) > **getConditionalFilterFiltered** ()=0  
*fetches the configuration of the conditional filter*
- virtual std::vector< [channel\\_t](#) > **getChannelList** ()=0  
*Fetches channels from the input file.*

## Public Member Functions inherited from [TimeTaggerBase](#)

- virtual unsigned int **getFence** (bool alloc\_fence=true)=0  
*Generate a new fence object, which validates the current configuration and the current time.*
- virtual bool **waitForFence** (unsigned int fence, int64\_t timeout=-1)=0  
*Wait for a fence in the data stream.*
- virtual bool **sync** (int64\_t timeout=-1)=0  
*Sync the timetagger pipeline, so that all started iterators and their enabled channels are ready.*
- virtual [channel\\_t](#) **getInvertedChannel** ([channel\\_t](#) channel)=0  
*get the falling channel id for a rising channel and vice versa*
- virtual bool **isUnusedChannel** ([channel\\_t](#) channel)=0  
*compares the provided channel with CHANNEL\_UNUSED*
- virtual void **runSynchronized** (const [IteratorCallbackMap](#) &callbacks, bool block=true)=0  
*Run synchronized callbacks for a list of iterators.*
- virtual std::string **getConfiguration** ()=0  
*Fetches the overall configuration status of the Time Tagger object.*
- virtual void **setInputDelay** ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void **setDelayHardware** ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual void **setDelaySoftware** ([channel\\_t](#) channel, [timestamp\\_t](#) delay)=0  
*set time delay on a channel*
- virtual [timestamp\\_t](#) **getInputDelay** ([channel\\_t](#) channel)=0  
*get time delay of a channel*

- virtual `timestamp_t getDelaySoftware (channel_t channel)=0`  
*get time delay of a channel*
- virtual `timestamp_t getDelayHardware (channel_t channel)=0`  
*get time delay of a channel*
- virtual `timestamp_t setDeadtime (channel_t channel, timestamp_t deadtime)=0`  
*set the deadtime between two edges on the same channel.*
- virtual `timestamp_t getDeadtime (channel_t channel)=0`  
*get the deadtime between two edges on the same channel.*
- virtual void `setTestSignal (channel_t channel, bool enabled)=0`  
*enable/disable internal test signal on a channel.*
- virtual void `setTestSignal (std::vector< channel_t > channel, bool enabled)=0`  
*enable/disable internal test signal on multiple channels.*
- virtual bool `getTestSignal (channel_t channel)=0`  
*fetch the status of the test signal generator*
- virtual void `setSoftwareClock (channel_t input_channel, double input_frequency=10e6, double averaging_↔ periods=1000, bool wait_until_locked=true)=0`  
*enables a software PLL to lock the time to an external clock*
- virtual void `disableSoftwareClock ()=0`  
*disabled the software PLL*
- virtual `SoftwareClockState getSoftwareClockState ()=0`  
*queries all state information of the software clock*
- virtual long long `getOverflows ()=0`  
*get overflow count*
- virtual void `clearOverflows ()=0`  
*clear overflow counter*
- virtual long long `getOverflowsAndClear ()=0`  
*get and clear overflow counter*

### Additional Inherited Members

### Public Types inherited from `TimeTaggerBase`

- typedef std::function< void(`IteratorBase` \*) `IteratorCallback`
- typedef std::map< `IteratorBase` \*, `IteratorCallback` > `IteratorCallbackMap`

### Protected Member Functions inherited from `TimeTaggerBase`

- `TimeTaggerBase ()`  
*abstract interface class*
- virtual `~TimeTaggerBase ()`  
*destructor*
- `TimeTaggerBase (const TimeTaggerBase &)=delete`
- `TimeTaggerBase & operator= (const TimeTaggerBase &)=delete`
- virtual std::shared\_ptr< `IteratorBaseListNode` > `addIterator (IteratorBase *it)=0`
- virtual void `freeIterator (IteratorBase *it)=0`
- virtual `channel_t getNewVirtualChannel ()=0`
- virtual void `freeVirtualChannel (channel_t channel)=0`
- virtual void `registerChannel (channel_t channel)=0`  
*register a FPGA channel.*
- virtual void `registerChannel (std::set< channel_t > channels)=0`

- virtual void `unregisterChannel` (`channel_t` channel)=0  
*release a previously registered channel.*
- virtual void `unregisterChannel` (std::set< `channel_t` > channels)=0
- virtual void `addChild` (`TimeTaggerBase` \*child)=0
- virtual void `removeChild` (`TimeTaggerBase` \*child)=0
- virtual void `release` ()=0

### 9.72.1 Detailed Description

virtual `TimeTagger` based on dump files

The `TimeTaggerVirtual` class represents a virtual Time Tagger. But instead of connecting to Swabian hardware, it replays all tags from a recorded file.

### 9.72.2 Member Function Documentation

#### 9.72.2.1 `clearConditionalFilter()`

```
virtual void TimeTaggerVirtual::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to `setConditionalFilter({},{})`

#### 9.72.2.2 `getChannelList()`

```
virtual std::vector< channel_t > TimeTaggerVirtual::getChannelList ( ) [pure virtual]
```

Fetches channels from the input file.

Returns

a vector of channels from the input file.

#### 9.72.2.3 `getConditionalFilterFiltered()`

```
virtual std::vector< channel_t > TimeTaggerVirtual::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see `setConditionalFilter`

#### 9.72.2.4 `getConditionalFilterTrigger()`

```
virtual std::vector< channel_t > TimeTaggerVirtual::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see `setConditionalFilter`

### 9.72.2.5 getReplaySpeed()

```
virtual double TimeTaggerVirtual::getReplaySpeed ( ) [pure virtual]
```

fetches the speed factor

Please see setReplaySpeed for more details.

#### Returns

the speed factor

### 9.72.2.6 replay()

```
virtual uint64_t TimeTaggerVirtual::replay (
    const std::string & file,
    timestamp_t begin = 0,
    timestamp_t duration = -1,
    bool queue = true ) [pure virtual]
```

replay a given dump file on the disc

This method adds the file to the replay queue. If the flag 'queue' is false, the current queue will be flushed and this file will be replayed immediately.

#### Parameters

<i>file</i>	the file to be replayed, must be encoded as UTF-8
<i>begin</i>	amount of ps to skip at the begin of the file. A negative time will generate a pause in the replay
<i>duration</i>	time period in ps of the file. -1 replays till the last tag
<i>queue</i>	flag if this file shall be queued

#### Returns

ID of the queued file

### 9.72.2.7 reset()

```
virtual void TimeTaggerVirtual::reset ( ) [pure virtual]
```

stops the all queued files and resets the [TimeTaggerVirtual](#) to its default settings

This method stops the current file, clears the replay queue and resets the [TimeTaggerVirtual](#) to its default settings.

### 9.72.2.8 setConditionalFilter()

```
virtual void TimeTaggerVirtual::setConditionalFilter (
    std::vector< channel_t > trigger,
    std::vector< channel_t > filtered ) [pure virtual]
```

configures the conditional filter

After each event on the trigger channels, one event per filtered channel will pass afterwards. This is handled in a very early stage in the pipeline, so all event limitations but the deadtime are suppressed. But the accuracy of the order of those events is low.

Refer the Manual for a description of this function.

#### Parameters

<i>trigger</i>	the channels that sets the condition
<i>filtered</i>	the channels that are filtered by the condition

### 9.72.2.9 setReplaySpeed()

```
virtual void TimeTaggerVirtual::setReplaySpeed (
    double speed ) [pure virtual]
```

configures the speed factor for the virtual tagger.

This method configures the speed factor of this virtual Time Tagger. A value of 1.0 will replay in real time. All values < 0.0 will replay the data as fast as possible, but stops at the end of all data. This is the default value.

#### Parameters

<i>speed</i>	ratio of the replay speed and the real time
--------------	---

### 9.72.2.10 stop()

```
virtual void TimeTaggerVirtual::stop ( ) [pure virtual]
```

stops the current and all queued files.

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

### 9.72.2.11 waitForCompletion()

```
virtual bool TimeTaggerVirtual::waitForCompletion (
    uint64_t ID = 0,
    int64_t timeout = -1 ) [pure virtual]
```

block the current thread until the replay finish

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

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

## Parameters

<i>ID</i>	selects which file to wait for
<i>timeout</i>	timeout in milliseconds

## Returns

true if the file is complete, false on timeout

The documentation for this class was generated from the following file:

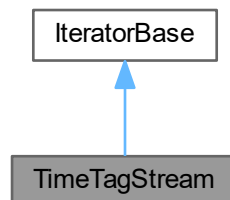
- [TimeTagger.h](#)

## 9.73 TimeTagStream Class Reference

access the time tag stream

```
#include <Iterators.h>
```

Inheritance diagram for TimeTagStream:



## Public Member Functions

- [TimeTagStream](#) ([TimeTaggerBase](#) \*[tagger](#), uint64\_t n\_max\_events, std::vector< [channel\\_t](#) > channels)  
*constructor of a [TimeTagStream](#) thread*
- [~TimeTagStream](#) ()
- uint64\_t [getCounts](#) ()  
*return the number of stored tags*
- [TimeTagStreamBuffer](#) [getData](#) ()  
*fetches all stored tags and clears the internal state*

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void [clear\\_impl](#) () override  
*clear [Iterator](#) state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- [std::unique\\_lock< std::mutex > getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- [template<typename T >](#)  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- [std::set< channel\\_t > channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase \\* tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t pre\\_capture\\_duration](#)  
*For internal use.*
- [std::atomic< bool > aborting](#)

### 9.73.1 Detailed Description

access the time tag stream

### 9.73.2 Constructor & Destructor Documentation

#### 9.73.2.1 TimeTagStream()

```
TimeTagStream::TimeTagStream (
    TimeTaggerBase * tagger,
    uint64_t n_max_events,
    std::vector< channel\_t > channels )
```

constructor of a [TimeTagStream](#) thread

Gives access to the time tag stream



## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>n_max_events</i>	maximum number of tags stored
<i>channels</i>	channels which are dumped to the file

**9.73.2.2 ~TimeTagStream()**

```
TimeTagStream::~~TimeTagStream ( )
```

**9.73.3 Member Function Documentation****9.73.3.1 clear\_impl()**

```
void TimeTagStream::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

**9.73.3.2 getCounts()**

```
uint64_t TimeTagStream::getCounts ( )
```

return the number of stored tags

**9.73.3.3 getData()**

```
TimeTagStreamBuffer TimeTagStream::getData ( )
```

fetches all stored tags and clears the internal state

**9.73.3.4 next\_impl()**

```
bool TimeTagStream::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp\_t begin_time,
    timestamp\_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

**Parameters**

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.74 TimeTagStreamBuffer Class Reference

return object for [TimeTagStream::getData](#)

```
#include <Iterators.h>
```

**Public Member Functions**

- [~TimeTagStreamBuffer](#) ()
- void [getOverflows](#) (std::function< unsigned char \*(size\_t)> array\_out)
- void [getChannels](#) (std::function< [channel\\_t](#) \*(size\_t)> array\_out)
- void [getTimestamps](#) (std::function< [timestamp\\_t](#) \*(size\_t)> array\_out)
- void [getMissedEvents](#) (std::function< unsigned short \*(size\_t)> array\_out)
- void [getEventTypes](#) (std::function< unsigned char \*(size\_t)> array\_out)

**Public Attributes**

- uint64\_t [size](#)
- bool [hasOverflows](#)
- [timestamp\\_t](#) [tStart](#)
- [timestamp\\_t](#) [tGetData](#)

### 9.74.1 Detailed Description

return object for [TimeTagStream::getData](#)

### 9.74.2 Constructor & Destructor Documentation

#### 9.74.2.1 ~TimeTagStreamBuffer()

```
TimeTagStreamBuffer::~TimeTagStreamBuffer ( )
```

## 9.74.3 Member Function Documentation

### 9.74.3.1 getChannels()

```
void TimeTagStreamBuffer::getChannels (
    std::function< channel\_t *(size_t)> array_out )
```

### 9.74.3.2 getEventTypes()

```
void TimeTagStreamBuffer::getEventTypes (
    std::function< unsigned char *(size_t)> array_out )
```

### 9.74.3.3 getMissedEvents()

```
void TimeTagStreamBuffer::getMissedEvents (
    std::function< unsigned short *(size_t)> array_out )
```

### 9.74.3.4 getOverflows()

```
void TimeTagStreamBuffer::getOverflows (
    std::function< unsigned char *(size_t)> array_out )
```

### 9.74.3.5 getTimestamps()

```
void TimeTagStreamBuffer::getTimestamps (
    std::function< timestamp\_t *(size_t)> array_out )
```

## 9.74.4 Member Data Documentation

### 9.74.4.1 hasOverflows

```
bool TimeTagStreamBuffer::hasOverflows
```

### 9.74.4.2 size

```
uint64_t TimeTagStreamBuffer::size
```

### 9.74.4.3 tGetData

```
timestamp\_t TimeTagStreamBuffer::tGetData
```

#### 9.74.4.4 tStart

`timestamp_t` TimeTagStreamBuffer::tStart

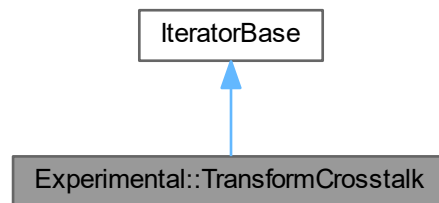
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.75 Experimental::TransformCrosstalk Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::TransformCrosstalk:



### Public Member Functions

- [TransformCrosstalk](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, [channel\\_t](#) relay\_input\_channel, double delay, double tau, bool copy=false)  
*Construct a transformation that will apply crosstalk effect between an input channel and a relay channel.*
- [~TransformCrosstalk](#) ()
- [channel\\_t](#) [getChannel](#) ()

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()

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

- void `abort ()`  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning ()`  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration ()`  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration ()`  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl (std::vector< Tag > &incoming_tags, timestamp_t begin_time, timestamp_t end_time)` override  
*update iterator state*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase (TimeTaggerBase *tagger, std::string base_type_="IteratorBase", std::string extra_info_="")`  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel (channel_t channel)`  
*register a channel*
- void `unregisterChannel (channel_t channel)`  
*unregister a channel*
- `channel_t getNewVirtualChannel ()`  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization ()`  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl ()`  
*clear `Iterator` state.*
- virtual void `on_start ()`  
*callback when the measurement class is started*
- virtual void `on_stop ()`  
*callback when the measurement class is stopped*
- void `lock ()`  
*acquire update lock*
- void `unlock ()`  
*release update lock*
- `OrderedBarrier::OrderInstance parallelize (OrderedPipeline &pipeline)`  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex > getLock ()`  
*acquire update lock*
- void `finish_running ()`  
*Callback for the measurement to stop itself.*
- void `checkForAbort ()`
- template<typename T >  
void `checkForAbort (T callback)`

## Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- `std::set< channel\_t > channels\_registered`  
*list of channels used by the iterator*
- `bool running`  
*running state of the iterator*
- `bool autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase * tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp\_t capture\_duration`  
*Duration the iterator has already processed data.*
- `timestamp\_t pre\_capture\_duration`  
*For internal use.*
- `std::atomic< bool > aborting`

## 9.75.1 Constructor & Destructor Documentation

### 9.75.1.1 TransformCrosstalk()

```
Experimental::TransformCrosstalk::TransformCrosstalk (
    TimeTaggerBase * tagger,
    channel\_t input_channel,
    channel\_t relay_input_channel,
    double delay,
    double tau,
    bool copy = false )
```

Construct a transformation that will apply crosstalk effect between an input channel and a relay channel.

#### Note

this measurement is a transformation, it will modify the input channel unless its copy parameter is set to true, in that case the modifications will be reflected on a virtual channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel to transform.
<i>relay_input_channel</i>	channel that causes the delays
<i>delay</i>	amount of delay triggered by relay channel.
<i>tau</i>	the decay after which an event of relay input channel has no effect anymore.
<i>copy</i>	tells if this transformation modifies the input or creates a new virtual channel with the transformation.

### 9.75.1.2 ~TransformCrosstalk()

```
Experimental::TransformCrosstalk::~~TransformCrosstalk ( )
```

## 9.75.2 Member Function Documentation

### 9.75.2.1 getChannel()

```
channel_t Experimental::TransformCrosstalk::getChannel ( )
```

### 9.75.2.2 next\_impl()

```
bool Experimental::TransformCrosstalk::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

#### Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

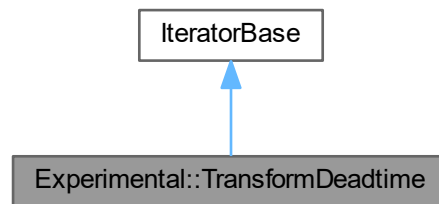
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.76 Experimental::TransformDeadtime Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for `Experimental::TransformDeadtime`:



### Public Member Functions

- `TransformDeadtime` (`TimeTaggerBase *tagger`, `channel_t input_channel`, double `deadtime`, bool `copy=false`)  
*Construct a transformation that will apply deadtime every event, filtering any events within the deadtime period.*
- `~TransformDeadtime` ()
- `channel_t getChannel` ()

### Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void `start` ()  
*Starts or continues data acquisition.*
- void `startFor` (`timestamp_t capture_duration`, bool `clear=true`)  
*Starts or continues the data acquisition for the given duration.*
- bool `waitUntilFinished` (`int64_t timeout=-1`)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- void `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- `timestamp_t getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- `std::string getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

### Protected Member Functions

- bool `next_impl` (`std::vector< Tag > &incoming_tags`, `timestamp_t begin_time`, `timestamp_t end_time`) override  
*update iterator state*



## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

## 9.76.1 Constructor & Destructor Documentation

### 9.76.1.1 TransformDeadtime()

```
Experimental::TransformDeadtime::TransformDeadtime (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    double deadtime,
    bool copy = false )
```

Construct a transformation that will apply deadtime every event, filtering any events within the deadtime period.

#### Note

this measurement is a transformation, it will modify the input channel unless its copy parameter is set to true, in that case the modifications will be reflected on a virtual channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel to transform.
<i>deadtime</i>	deadtime in seconds.
<i>copy</i>	tells if this transformation modifies the input or creates a new virtual channel with the transformation.

### 9.76.1.2 ~TransformDeadtime()

```
Experimental::TransformDeadtime::~~TransformDeadtime ( )
```

## 9.76.2 Member Function Documentation

### 9.76.2.1 getChannel()

```
channel_t Experimental::TransformDeadtime::getChannel ( )
```

### 9.76.2.2 next\_impl()

```
bool Experimental::TransformDeadtime::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

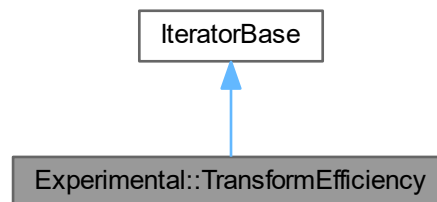
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.77 Experimental::TransformEfficiency Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::TransformEfficiency:



## Public Member Functions

- [TransformEfficiency](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, double efficiency, bool copy=false, [int32\\_t](#) seed=-1)  
*Construct a transformation that will apply an efficiency filter to an specified channel. An efficiency filter will drop events based on an efficiency value. A perfect efficiency of 1.0 won't drop any events, an efficiency of 0.5 will drop half the events.*
- [~TransformEfficiency](#) ()
- [channel\\_t](#) getChannel ()

## Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) (TimeTaggerBase \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) (channel\_t channel)  
*register a channel*
- void [unregisterChannel](#) (channel\_t channel)  
*unregister a channel*
- channel\_t [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [clear\\_impl](#) ()  
*clear [Iterator](#) state.*
- virtual void [on\\_start](#) ()  
*callback when the measurement class is started*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()

- acquire update lock*
- void [unlock](#) ()
- release update lock*
- [OrderedBarrier::OrderInstance parallelize](#) ([OrderedPipeline](#) &pipeline)
- release lock and continue work in parallel*
- [std::unique\\_lock< std::mutex > getLock](#) ()
- acquire update lock*
- void [finish\\_running](#) ()
- Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- [template<typename T > void checkForAbort](#) (T callback)

### Additional Inherited Members

### Protected Attributes inherited from [IteratorBase](#)

- [std::set< channel\\_t > channels\\_registered](#)
- list of channels used by the iterator*
- bool [running](#)
- running state of the iterator*
- bool [autostart](#)
- Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase \\* tagger](#)
- Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t capture\\_duration](#)
- Duration the iterator has already processed data.*
- [timestamp\\_t pre\\_capture\\_duration](#)
- For internal use.*
- [std::atomic< bool > aborting](#)

## 9.77.1 Constructor & Destructor Documentation

### 9.77.1.1 TransformEfficiency()

```
Experimental::TransformEfficiency::TransformEfficiency (
    TimeTaggerBase \* tagger,
    channel\_t input\_channel,
    double efficiency,
    bool copy = false,
    int32\_t seed = -1 )
```

Construct a transformation that will apply an efficiency filter to an specified channel. An efficiency filter will drop events based on an efficiency value. A perfect efficiency of 1.0 won't drop any events, an efficiency of 0.5 will drop half the events.

#### Note

this measurement is a transformation, it will modify the input channel unless its copy parameter is set to true, in that case the modifications will be reflected on a virtual channel.

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel to be filtered.
<i>efficiency</i>	efficiency of the transformation. a 0.5 efficiency will drop half the events. A 1.0 won't drop any.
<i>copy</i>	tells if this transformation modifies the input or creates a new virtual channel with the transformation.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

9.77.1.2 `~TransformEfficiency()`

```
Experimental::TransformEfficiency::~~TransformEfficiency ( )
```

## 9.77.2 Member Function Documentation

9.77.2.1 `getChannel()`

```
channel_t Experimental::TransformEfficiency::getChannel ( )
```

9.77.2.2 `next_impl()`

```
bool Experimental::TransformEfficiency::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

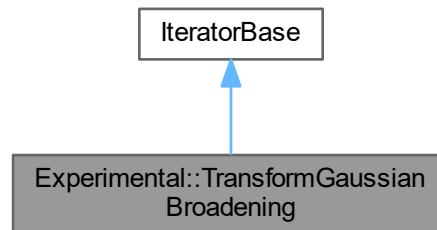
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.78 Experimental::TransformGaussianBroadening Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::TransformGaussianBroadening:



### Public Member Functions

- [TransformGaussianBroadening](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, double standard\_deviation, bool copy=false, [int32\\_t](#) seed=-1)  
Construct a transformation that will apply gaussian brodening to each event in an specified channel.
- [~TransformGaussianBroadening](#) ()
- [channel\\_t](#) getChannel ()

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
destructor, will unregister from the Time Tagger prior finalization.
- void [start](#) ()  
Starts or continues data acquisition.
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
Starts or continues the data acquisition for the given duration.
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).
- void [stop](#) ()  
After calling this method, the measurement will stop processing incoming tags.
- void [clear](#) ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- void [abort](#) ()  
Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.
- bool [isRunning](#) ()  
Returns True if the measurement is collecting the data.
- [timestamp\\_t](#) getCaptureDuration ()  
Total capture duration since the measurement creation or last call to [clear\(\)](#).
- std::string [getConfiguration](#) ()  
Fetches the overall configuration status of the measurement.

### Protected Member Functions

- bool `next_impl` (std::vector< `Tag` > &incoming\_tags, `timestamp_t` begin\_time, `timestamp_t` end\_time) override  
*update iterator state*

### Protected Member Functions inherited from `IteratorBase`

- `IteratorBase` (`TimeTaggerBase` \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void `registerChannel` (`channel_t` channel)  
*register a channel*
- void `unregisterChannel` (`channel_t` channel)  
*unregister a channel*
- `channel_t` `getNewVirtualChannel` ()  
*allocate a new virtual output channel for this iterator*
- void `finishInitialization` ()  
*method to call after finishing the initialization of the measurement*
- virtual void `clear_impl` ()  
*clear `Iterator` state.*
- virtual void `on_start` ()  
*callback when the measurement class is started*
- virtual void `on_stop` ()  
*callback when the measurement class is stopped*
- void `lock` ()  
*acquire update lock*
- void `unlock` ()  
*release update lock*
- `OrderedBarrier::OrderInstance` `parallelize` (`OrderedPipeline` &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > `getLock` ()  
*acquire update lock*
- void `finish_running` ()  
*Callback for the measurement to stop itself.*
- void `checkForAbort` ()
- template<typename T >  
void `checkForAbort` (T callback)

### Additional Inherited Members

### Protected Attributes inherited from `IteratorBase`

- std::set< `channel_t` > `channels_registered`  
*list of channels used by the iterator*
- bool `running`  
*running state of the iterator*
- bool `autostart`  
*Condition if this measurement shall be started by the finishInitialization callback.*
- `TimeTaggerBase` \* `tagger`  
*Pointer to the corresponding Time Tagger object.*
- `timestamp_t` `capture_duration`  
*Duration the iterator has already processed data.*
- `timestamp_t` `pre_capture_duration`  
*For internal use.*
- std::atomic< bool > `aborting`



## 9.78.1 Constructor & Destructor Documentation

### 9.78.1.1 TransformGaussianBroadening()

```
Experimental::TransformGaussianBroadening::TransformGaussianBroadening (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    double standard_deviation,
    bool copy = false,
    int32_t seed = -1 )
```

Construct a transformation that will apply gaussian brodening to each event in an specified channel.

#### Note

this measurement is a transformation, it will modify the input channel unless its copy parameter is set to true, in that case the modifications will be reflected on a virtual channel.

-2 broadening will be limited to 5 times the standard deviation.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>input_channel</i>	channel to be transformed.
<i>standard_deviation</i>	gaussian standard deviation which will affect the broadening
<i>copy</i>	tells if this transformation modifies the input or creates a new virtual channel with the transformation.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.78.1.2 ~TransformGaussianBroadening()

```
Experimental::TransformGaussianBroadening::~~TransformGaussianBroadening ( )
```

## 9.78.2 Member Function Documentation

### 9.78.2.1 getChannel()

```
channel_t Experimental::TransformGaussianBroadening::getChannel ( )
```

### 9.78.2.2 next\_impl()

```
bool Experimental::TransformGaussianBroadening::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each [Iterator](#) must implement the [next\\_impl\(\)](#) method. The [next\\_impl\(\)](#) function is guarded by the update lock.

The backend delivers each [Tag](#) on each registered channel to this callback function.

## Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

## Returns

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

The documentation for this class was generated from the following file:

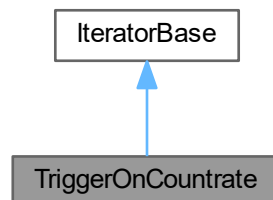
- [Iterators.h](#)

## 9.79 TriggerOnCountrate Class Reference

Inject trigger events when exceeding or falling below a given count rate within a rolling time window.

```
#include <Iterators.h>
```

Inheritance diagram for TriggerOnCountrate:



## Public Member Functions

- [TriggerOnCountrate](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) input\_channel, double reference\_countrate, double hysteresis, [timestamp\\_t](#) time\_window)  
*constructor of a [TriggerOnCountrate](#)*
- [~TriggerOnCountrate](#) ()
- [channel\\_t](#) getChannelAbove ()  
*Get the channel number of the above channel.*
- [channel\\_t](#) getChannelBelow ()  
*Get the channel number of the below channel.*
- `std::vector< channel\_t >` getChannels ()  
*Get both virtual channel numbers: [[getChannelAbove\(\)](#), [getChannelBelow\(\)](#)].*

- 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.*
- double `getCurrentCountrate` ()  
*Get the current count rate averaged within the `time_window`.*
- bool `injectCurrentState` ()  
*Emit a time-tag into the respective channel according to the current state.*

## Public Member Functions inherited from `IteratorBase`

- virtual `~IteratorBase` ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void `start` ()  
*Starts or continues data acquisition.*
- void `startFor` (timestamp\_t capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool `waitUntilFinished` (int64\_t timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with `startFor()`.*
- void `stop` ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void `clear` ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void `abort` ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool `isRunning` ()  
*Returns True if the measurement is collecting the data.*
- timestamp\_t `getCaptureDuration` ()  
*Total capture duration since the measurement creation or last call to `clear()`.*
- std::string `getConfiguration` ()  
*Fetches the overall configuration status of the measurement.*

## Protected Member Functions

- bool `next_impl` (std::vector< Tag > &incoming\_tags, timestamp\_t begin\_time, timestamp\_t end\_time) override  
*update iterator state*
- void `on_start` () override  
*callback when the measurement class is started*
- void `clear_impl` () override  
*clear `Iterator` state.*

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
*Standard constructor, which will register with the Time Tagger backend.*
- void [registerChannel](#) ([channel\\_t](#) channel)  
*register a channel*
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
*unregister a channel*
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
*allocate a new virtual output channel for this iterator*
- void [finishInitialization](#) ()  
*method to call after finishing the initialization of the measurement*
- virtual void [on\\_stop](#) ()  
*callback when the measurement class is stopped*
- void [lock](#) ()  
*acquire update lock*
- void [unlock](#) ()  
*release update lock*
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- std::unique\_lock< std::mutex > [getLock](#) ()  
*acquire update lock*
- void [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

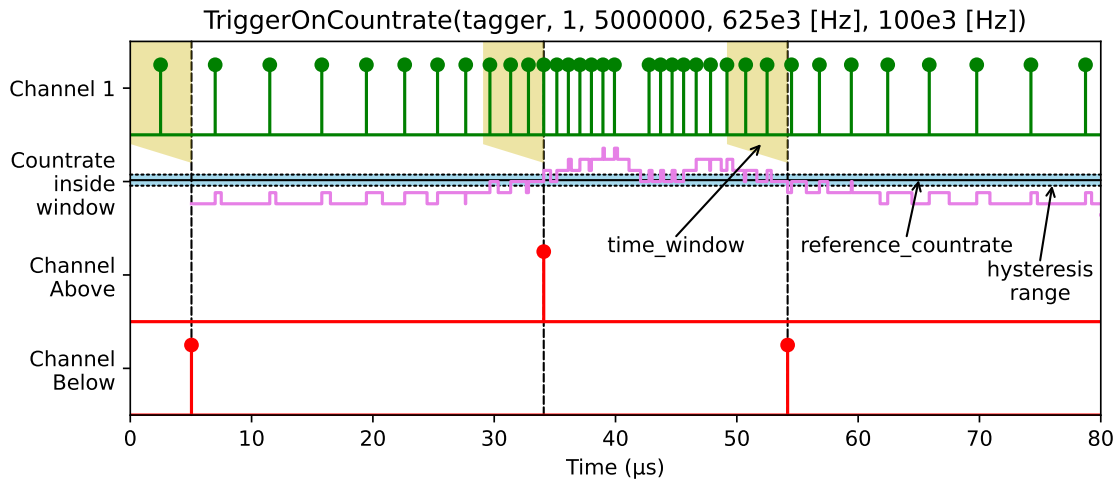
## Additional Inherited Members

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
*list of channels used by the iterator*
- bool [running](#)  
*running state of the iterator*
- bool [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- std::atomic< bool > [aborting](#)

### 9.79.1 Detailed Description

Inject trigger events when exceeding or falling below a given count rate within a rolling time window.



Measures the count rate inside a rolling time window and emits tags when a given `reference_countrate` is crossed. A `TriggerOnCountrate` object provides two virtual channels: The `above` channel is triggered when the count rate exceeds the threshold (transition from `below` to `above`). The `below` channel is triggered when the count rate falls below the threshold (transition from `above` to `below`). To avoid the emission of multiple trigger tags in the transition area, the `hysteresis` count rate modifies the threshold with respect to the transition direction: An event in the `above` channel will be triggered when the channel is in the `below` state and rises to `reference_countrate + hysteresis` or above. Vice versa, the `below` channel fires when the channel is in the `above` state and falls to the limit of `reference_countrate - hysteresis` or below.

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

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

### 9.79.2 Constructor & Destructor Documentation

#### 9.79.2.1 TriggerOnCountrate()

```
TriggerOnCountrate::TriggerOnCountrate (
    TimeTaggerBase * tagger,
    channel_t input_channel,
    double reference_countrate,
    double hysteresis,
    timestamp_t time_window )
```

constructor of a `TriggerOnCountrate`

#### Parameters

<i>tagger</i>	Reference to a <code>TimeTagger</code> object.
<i>input_channel</i>	Channel number of the channel whose count rate will control the trigger channels.

## Parameters

<i>reference_countrate</i>	The reference count rate in Hz that separates the <code>above</code> range from the <code>below</code> range.
<i>hysteresis</i>	The threshold count rate in Hz for transitioning to the <code>above</code> threshold state is <code>countrate &gt;= reference_countrate + hysteresis</code> , whereas it is <code>countrate &lt;= reference_countrate - hysteresis</code> for transitioning to the <code>below</code> threshold state. The hysteresis avoids the emission of multiple trigger tags upon a single transition.
<i>time_window</i>	Rolling time window size in ps. The count rate is analyzed within this time window and compared to the threshold count rate.

9.79.2.2 `~TriggerOnCountrate()`

```
TriggerOnCountrate::~TriggerOnCountrate ( )
```

## 9.79.3 Member Function Documentation

9.79.3.1 `clear_impl()`

```
void TriggerOnCountrate::clear_impl ( ) [override], [protected], [virtual]
```

clear [Iterator](#) state.

Each [Iterator](#) should implement the [clear\\_impl\(\)](#) method to reset its internal state. The [clear\\_impl\(\)](#) function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

9.79.3.2 `getChannelAbove()`

```
channel_t TriggerOnCountrate::getChannelAbove ( )
```

Get the channel number of the `above` channel.

9.79.3.3 `getChannelBelow()`

```
channel_t TriggerOnCountrate::getChannelBelow ( )
```

Get the channel number of the `below` channel.

9.79.3.4 `getChannels()`

```
std::vector< channel_t > TriggerOnCountrate::getChannels ( )
```

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

### 9.79.3.5 getCurrentCountrate()

```
double TriggerOnCountrate::getCurrentCountrate ( )
```

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

### 9.79.3.6 injectCurrentState()

```
bool TriggerOnCountrate::injectCurrentState ( )
```

Emit a time-tag into the respective channel according to the current state.

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

### 9.79.3.7 isAbove()

```
bool TriggerOnCountrate::isAbove ( )
```

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

### 9.79.3.8 isBelow()

```
bool TriggerOnCountrate::isBelow ( )
```

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

### 9.79.3.9 next\_impl()

```
bool TriggerOnCountrate::next_impl (
    std::vector< Tag > & incoming_tags,
    timestamp_t begin_time,
    timestamp_t end_time ) [override], [protected], [virtual]
```

update iterator state

Each `Iterator` must implement the `next_impl()` method. The `next_impl()` function is guarded by the update lock.

The backend delivers each `Tag` on each registered channel to this callback function.

#### Parameters

<i>incoming_tags</i>	block of events
<i>begin_time</i>	earliest event in the block
<i>end_time</i>	begin_time of the next block, not including in this block

**Returns**

true if the content of this block was modified, false otherwise

Implements [IteratorBase](#).

**9.79.3.10 on\_start()**

```
void TriggerOnCountrate::on_start ( ) [override], [protected], [virtual]
```

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

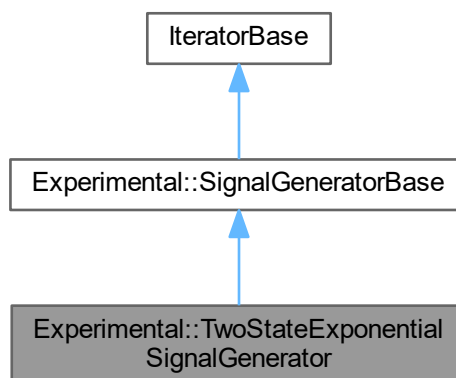
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.80 Experimental::TwoStateExponentialSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::TwoStateExponentialSignalGenerator:

**Public Member Functions**

- [TwoStateExponentialSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, double excitation\_time, double life\_time, [channel\\_t](#) base\_channel=CHANNEL\_UNUSED, [int32\\_t](#) seed=-1)  
Construct a two-state exponential event channel.
- [~TwoStateExponentialSignalGenerator](#) ()



**Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)**

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=[CHANNEL\\_UNUSED](#))
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) [getChannel](#) ()  
*the new virtual channel*

**Public Member Functions inherited from [IteratorBase](#)**

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) [capture\\_duration](#), bool [clear](#)=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) [timeout](#)=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()  
*After calling this method, the measurement will stop processing incoming tags.*
- void [clear](#) ()  
*Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.*
- void [abort](#) ()  
*Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.*
- bool [isRunning](#) ()  
*Returns True if the measurement is collecting the data.*
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
*Total capture duration since the measurement creation or last call to [clear\(\)](#).*
- std::string [getConfiguration](#) ()  
*Fetches the overall configuration status of the measurement.*

**Protected Member Functions**

- void [initialize](#) ([timestamp\\_t](#) [initial\\_time](#)) override
- [timestamp\\_t](#) [get\\_next](#) () override
- void [on\\_restart](#) ([timestamp\\_t](#) [restart\\_time](#)) override

**Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)**

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) [begin\\_time](#), [timestamp\\_t](#) [end\\_time](#)) override  
*update iterator state*
- void [on\\_stop](#) () override  
*callback when the measurement class is stopped*
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool [is\\_finished](#))

## Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
Standard constructor, which will register with the Time Tagger backend.
- void [registerChannel](#) ([channel\\_t](#) channel)  
register a channel
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
unregister a channel
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
allocate a new virtual output channel for this iterator
- void [finishInitialization](#) ()  
method to call after finishing the initialization of the measurement
- virtual void [clear\\_impl](#) ()  
clear [Iterator](#) state.
- virtual void [on\\_start](#) ()  
callback when the measurement class is started
- void [lock](#) ()  
acquire update lock
- void [unlock](#) ()  
release update lock
- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
release lock and continue work in parallel
- std::unique\_lock< std::mutex > [getLock](#) ()  
acquire update lock
- void [finish\\_running](#) ()  
Callback for the measurement to stop itself.
- void [checkForAbort](#) ()
- template<typename T >  
void [checkForAbort](#) (T callback)

## Additional Inherited Members

## Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- std::unique\_ptr< [SignalGeneratorBaseImpl](#) > impl

## Protected Attributes inherited from [IteratorBase](#)

- std::set< [channel\\_t](#) > [channels\\_registered](#)  
list of channels used by the iterator
- bool [running](#)  
running state of the iterator
- bool [autostart](#)  
Condition if this measurement shall be started by the finishInitialization callback.
- [TimeTaggerBase](#) \* [tagger](#)  
Pointer to the corresponding Time Tagger object.
- [timestamp\\_t](#) [capture\\_duration](#)  
Duration the iterator has already processed data.
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
For internal use.
- std::atomic< bool > [aborting](#)

## 9.80.1 Constructor & Destructor Documentation

### 9.80.1.1 TwoStateExponentialSignalGenerator()

```
Experimental::TwoStateExponentialSignalGenerator::TwoStateExponentialSignalGenerator (
    TimeTaggerBase * tagger,
    double excitation_time,
    double life_time,
    channel_t base_channel = CHANNEL_UNUSED,
    int32_t seed = -1 )
```

Construct a two-state exponential event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>excitation_time</i>	excitation time in seconds.
<i>life_time</i>	life time of the excited state in seconds
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.80.1.2 ~TwoStateExponentialSignalGenerator()

```
Experimental::TwoStateExponentialSignalGenerator::~~TwoStateExponentialSignalGenerator ( )
```

## 9.80.2 Member Function Documentation

### 9.80.2.1 get\_next()

```
timestamp_t Experimental::TwoStateExponentialSignalGenerator::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.80.2.2 initialize()

```
void Experimental::TwoStateExponentialSignalGenerator::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.80.2.3 on\_restart()

```
void Experimental::TwoStateExponentialSignalGenerator::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

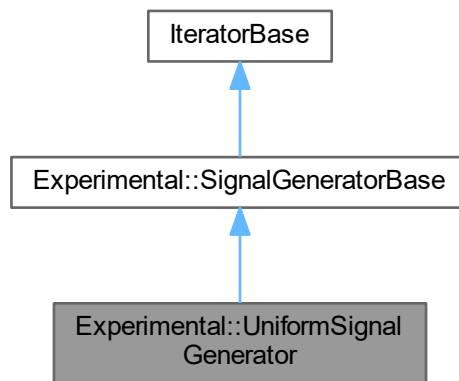
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.81 Experimental::UniformSignalGenerator Class Reference

```
#include <Iterators.h>
```

Inheritance diagram for Experimental::UniformSignalGenerator:



### Public Member Functions

- [UniformSignalGenerator](#) ([TimeTaggerBase](#) \*tagger, [timestamp\\_t](#) upper\_bound, [timestamp\\_t](#) lower\_bound=1, [channel\\_t](#) base\_channel=CHANNEL\_UNUSED, [int32\\_t](#) seed=-1)  
*Construct a random uniform event channel.*
- [~UniformSignalGenerator](#) ()

### Public Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- [SignalGeneratorBase](#) ([TimeTaggerBase](#) \*tagger, [channel\\_t](#) base\_channel=CHANNEL\_UNUSED)
- [~SignalGeneratorBase](#) ()
- [channel\\_t](#) getChannel ()  
*the new virtual channel*

### Public Member Functions inherited from [IteratorBase](#)

- virtual [~IteratorBase](#) ()  
*destructor, will unregister from the Time Tagger prior finalization.*
- void [start](#) ()  
*Starts or continues data acquisition.*
- void [startFor](#) ([timestamp\\_t](#) capture\_duration, bool clear=true)  
*Starts or continues the data acquisition for the given duration.*
- bool [waitUntilFinished](#) ([int64\\_t](#) timeout=-1)  
*Blocks the execution until the measurement has finished. Can be used with [startFor\(\)](#).*
- void [stop](#) ()

After calling this method, the measurement will stop processing incoming tags.

- void [clear](#) ()  
Discards accumulated measurement data, initializes the data buffer with zero values, and resets the state to the initial state.
- void [abort](#) ()  
Immediately aborts the measurement, discarding accumulated measurement data, and resets the state to the initial state.
- bool [isRunning](#) ()  
Returns True if the measurement is collecting the data.
- [timestamp\\_t](#) [getCaptureDuration](#) ()  
Total capture duration since the measurement creation or last call to [clear\(\)](#).
- std::string [getConfiguration](#) ()  
Fetches the overall configuration status of the measurement.

### Protected Member Functions

- void [initialize](#) ([timestamp\\_t](#) initial\_time) override
- [timestamp\\_t](#) [get\\_next](#) () override
- void [on\\_restart](#) ([timestamp\\_t](#) restart\_time) override

### Protected Member Functions inherited from [Experimental::SignalGeneratorBase](#)

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) begin\_time, [timestamp\\_t](#) end\_time) override  
update iterator state
- void [on\\_stop](#) () override  
callback when the measurement class is stopped
- bool [isProcessingFinished](#) ()
- void [set\\_processing\\_finished](#) (bool is\_finished)

### Protected Member Functions inherited from [IteratorBase](#)

- [IteratorBase](#) ([TimeTaggerBase](#) \*tagger, std::string base\_type\_="IteratorBase", std::string extra\_info\_="")  
Standard constructor, which will register with the Time Tagger backend.
- void [registerChannel](#) ([channel\\_t](#) channel)  
register a channel
- void [unregisterChannel](#) ([channel\\_t](#) channel)  
unregister a channel
- [channel\\_t](#) [getNewVirtualChannel](#) ()  
allocate a new virtual output channel for this iterator
- void [finishInitialization](#) ()  
method to call after finishing the initialization of the measurement
- virtual void [clear\\_impl](#) ()  
clear *Iterator* state.
- virtual void [on\\_start](#) ()  
callback when the measurement class is started
- void [lock](#) ()  
acquire update lock
- void [unlock](#) ()  
release update lock

- [OrderedBarrier::OrderInstance](#) [parallelize](#) ([OrderedPipeline](#) &pipeline)  
*release lock and continue work in parallel*
- `std::unique_lock< std::mutex >` [getLock](#) ()  
*acquire update lock*
- `void` [finish\\_running](#) ()  
*Callback for the measurement to stop itself.*
- `void` [checkForAbort](#) ()
- `template<typename T >`  
`void` [checkForAbort](#) (T callback)

### Additional Inherited Members

### Protected Attributes inherited from [Experimental::SignalGeneratorBase](#)

- `std::unique_ptr<` [SignalGeneratorBaseImpl](#) `>` [impl](#)

### Protected Attributes inherited from [IteratorBase](#)

- `std::set<` [channel\\_t](#) `>` [channels\\_registered](#)  
*list of channels used by the iterator*
- `bool` [running](#)  
*running state of the iterator*
- `bool` [autostart](#)  
*Condition if this measurement shall be started by the finishInitialization callback.*
- [TimeTaggerBase](#) \* [tagger](#)  
*Pointer to the corresponding Time Tagger object.*
- [timestamp\\_t](#) [capture\\_duration](#)  
*Duration the iterator has already processed data.*
- [timestamp\\_t](#) [pre\\_capture\\_duration](#)  
*For internal use.*
- `std::atomic< bool >` [aborting](#)

## 9.81.1 Constructor & Destructor Documentation

### 9.81.1.1 [UniformSignalGenerator](#)()

```
Experimental::UniformSignalGenerator::UniformSignalGenerator (
    TimeTaggerBase * tagger,
    timestamp\_t upper\_bound,
    timestamp\_t lower\_bound = 1,
    channel\_t base\_channel = CHANNEL\_UNUSED,
    int32\_t seed = -1 )
```

Construct a random uniform event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>upper_bound</i>	Max possible offset of event generated compared to latest.
<i>lower_bound</i>	Min possible offset of event generated, must be higher than 0.
<i>base_channel</i>	base channel to which this signal will be added. If unused, a new channel will be created.
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.81.1.2 ~UniformSignalGenerator()

```
Experimental::UniformSignalGenerator::~~UniformSignalGenerator ( )
```

## 9.81.2 Member Function Documentation

### 9.81.2.1 get\_next()

```
timestamp_t Experimental::UniformSignalGenerator::get_next ( ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.81.2.2 initialize()

```
void Experimental::UniformSignalGenerator::initialize (
    timestamp_t initial_time ) [override], [protected], [virtual]
```

Implements [Experimental::SignalGeneratorBase](#).

### 9.81.2.3 on\_restart()

```
void Experimental::UniformSignalGenerator::on_restart (
    timestamp_t restart_time ) [override], [protected], [virtual]
```

Reimplemented from [Experimental::SignalGeneratorBase](#).

The documentation for this class was generated from the following file:

- [Iterators.h](#)





# Chapter 10

## File Documentation

### 10.1 Iterators.h File Reference

```
#include <algorithm>
#include <array>
#include <assert.h>
#include <atomic>
#include <complex>
#include <deque>
#include <fstream>
#include <functional>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <memory>
#include <mutex>
#include <queue>
#include <random>
#include <set>
#include <stdint.h>
#include <stdio.h>
#include <unordered_map>
#include <vector>
#include "TimeTagger.h"
Include dependency graph for Iterators.h:
```



### Classes

- class [FastBinning](#)  
*Helper class for fast division with a constant divisor.*
- class [Combiner](#)  
*Combine some channels in a virtual channel which has a tick for each tick in the input channels.*
- class [CountBetweenMarkers](#)

- a simple counter where external marker signals determine the bins*
- class [CounterData](#)
  - Helper object as return value for [Counter::getDataObject](#).*
- class [Counter](#)
  - a simple counter on one or more channels*
- class [Coincidences](#)
  - a coincidence monitor for many channel groups*
- class [Coincidence](#)
  - a coincidence monitor for one channel group*
- class [Countrate](#)
  - count rate on one or more channels*
- class [DelayedChannel](#)
  - a simple delayed queue*
- class [TriggerOnCountrate](#)
  - Inject trigger events when exceeding or falling below a given count rate within a rolling time window.*
- class [GatedChannel](#)
  - An input channel is gated by a gate channel.*
- class [FrequencyMultiplier](#)
  - The signal of an input channel is scaled up to a higher frequency according to the multiplier passed as a parameter.*
- class [Iterator](#)
  - a deprecated simple event queue*
- class [TimeTagStreamBuffer](#)
  - return object for [TimeTagStream::getData](#)*
- class [TimeTagStream](#)
  - access the time tag stream*
- class [Dump](#)
  - dump all time tags to a file*
- class [StartStop](#)
  - simple start-stop measurement*
- class [TimeDifferences](#)
  - Accumulates the time differences between clicks on two channels in one or more histograms.*
- class [Histogram2D](#)
  - A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.*
- class [HistogramND](#)
  - A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.*
- class [TimeDifferencesND](#)
  - Accumulates the time differences between clicks on two channels in a multi-dimensional histogram.*
- class [Histogram](#)
  - Accumulate time differences into a histogram.*
- class [FrequencyCounterData](#)
- class [FrequencyCounter](#)
  - Calculate the phase of multiple channels at equidistant sampling points.*
- class [HistogramLogBinsData](#)
  - Helper object as return value for [HistogramLogBins::getDataObject](#).*
- struct [ChannelGate](#)
- class [HistogramLogBins](#)
  - Accumulate time differences into a histogram with logarithmic increasing bin sizes.*
- class [Correlation](#)
  - Auto- and Cross-correlation measurement.*
- struct [Event](#)
  - Object for the return value of [Scope::getData](#).*

- class [Scope](#)  
*a scope measurement*
- class [SynchronizedMeasurements](#)  
*start, stop and clear several measurements synchronized*
- class [ConstantFractionDiscriminator](#)  
*a virtual CFD implementation which returns the mean time between a rising and a falling pair of edges*
- class [FileWriter](#)  
*compresses and stores all time tags to a file*
- class [FileReader](#)  
*Reads tags from the disk files, which has been created by [FileWriter](#).*
- class [EventGenerator](#)  
*Generate predefined events in a virtual channel relative to a trigger event.*
- class [Combinations](#)
- class [CustomMeasurementBase](#)  
*Helper class for custom measurements in Python and C#.*
- class [FlimAbstract](#)  
*Interface for FLIM measurements, [Flim](#) and [FlimBase](#) classes inherit from it.*
- class [FlimBase](#)  
*basic measurement, containing a minimal set of features for efficiency purposes*
- class [FlimFrameInfo](#)  
*object for storing the state of [Flim::getCurrentFrameEx](#)*
- class [Flim](#)  
*Fluorescence lifetime imaging.*
- class [Sampler](#)  
*a triggered sampling measurement*
- class [SyntheticSingleTag](#)  
*synthetic trigger timetag generator.*
- class [FrequencyStabilityData](#)  
*return data object for [FrequencyStability::getData](#).*
- class [FrequencyStability](#)  
*Allan deviation (and related metrics) calculator.*
- class [Experimental::PulsePerSecondData](#)  
*Helper object as return value for [PulsePerSecondMonitor::getDataObject](#).*
- class [Experimental::PulsePerSecondMonitor](#)  
*Monitors the synchronicity of 1 pulse per second (PPS) signals.*
- class [Experimental::SignalGeneratorBase](#)
- class [Experimental::PhotonGenerator](#)
- class [Experimental::DlsSignalGenerator](#)
- class [Experimental::FcsSignalGenerator](#)
- class [Experimental::UniformSignalGenerator](#)
- class [Experimental::GaussianSignalGenerator](#)
- class [Experimental::OscillatorSimulation](#)
- class [Experimental::TwoStateExponentialSignalGenerator](#)
- class [Experimental::MarkovProcessGenerator](#)
- class [Experimental::ExponentialSignalGenerator](#)
- class [Experimental::GammaSignalGenerator](#)
- class [Experimental::PatternSignalGenerator](#)
- class [Experimental::SimSignalSplitter](#)
- class [Experimental::TransformEfficiency](#)
- class [Experimental::TransformGaussianBroadening](#)
- class [Experimental::TransformDeadtime](#)
- class [Experimental::TransformCrosstalk](#)

- class [Experimental::SimDetector](#)
- class [Experimental::SimLifetime](#)
- class [Experimental::PhotonNumber](#)

*Photon number resolution.*

## Namespaces

- namespace [Experimental](#)

*Namespace for features, which are still in development and are likely to change.*

## Macros

- `#define BINNING\_TEMPLATE\_HELPER(fun_name, binner, ...)`

*[FastBinning](#) caller helper.*

## Enumerations

- enum class [CoincidenceTimestamp](#) : `uint32_t` { [Last](#) = 0 , [Average](#) = 1 , [First](#) = 2 , [ListedFirst](#) = 3 }  
*type of timestamp for the [Coincidence](#) virtual channel (Last, Average, First, ListedFirst)*
- enum class [GatedChannelInitial](#) : `uint32_t` { [Closed](#) = 0 , [Open](#) = 1 }  
*Initial state of the gate of a [GatedChannel](#) (Closed, Open)*
- enum [State](#) { [UNKNOWN](#) , [HIGH](#) , [LOW](#) }  
*Input state in the return object of [Scope](#).*

## 10.1.1 Macro Definition Documentation

### 10.1.1.1 BINNING\_TEMPLATE\_HELPER

```
#define BINNING_TEMPLATE_HELPER(
    fun_name,
    binner,
    ... )
```

#### Value:

```
switch (binner.getMode()) {
case FastBinning::Mode::ConstZero:
    fun_name<FastBinning::Mode::ConstZero>(__VA_ARGS__);
    break;
case FastBinning::Mode::Dividend:
    fun_name<FastBinning::Mode::Dividend>(__VA_ARGS__);
    break;
case FastBinning::Mode::PowerOfTwo:
    fun_name<FastBinning::Mode::PowerOfTwo>(__VA_ARGS__);
    break;
case FastBinning::Mode::FixedPoint\_32:
    fun_name<FastBinning::Mode::FixedPoint\_32>(__VA_ARGS__);
}
```

```

    break;
  case FastBinning::Mode::FixedPoint_64:
    fun_name<FastBinning::Mode::FixedPoint_64>(__VA_ARGS__);
    break;
  case FastBinning::Mode::Divide_32:
    fun_name<FastBinning::Mode::Divide_32>(__VA_ARGS__);
    break;
  case FastBinning::Mode::Divide_64:
    fun_name<FastBinning::Mode::Divide_64>(__VA_ARGS__);
    break;
}

```

[FastBinning](#) caller helper.

## 10.1.2 Enumeration Type Documentation

### 10.1.2.1 CoincidenceTimestamp

```
enum class CoincidenceTimestamp : uint32_t [strong]
```

type of timestamp for the [Coincidence](#) virtual channel (Last, Average, First, ListedFirst)

Enumerator

Last	time of the last event completing the coincidence (fastest option - default)
Average	average time of all tags completing the coincidence
First	time of the first event received of the coincidence
ListedFirst	time of the first channel of the list with which the <a href="#">Coincidence</a> was initialized

### 10.1.2.2 GatedChannelInitial

```
enum class GatedChannelInitial : uint32_t [strong]
```

Initial state of the gate of a [GatedChannel](#) (Closed, Open)

Enumerator

Closed	the gate is closed initially (default)
Open	the gate is open initially

### 10.1.2.3 State

```
enum State
```

Input state in the return object of [Scope](#).

## Enumerator

UNKNOWN	
HIGH	
LOW	

## 10.2 Iterators.h

[Go to the documentation of this file.](#)

```

00001  /*
00002  This file is part of Time Tagger software defined digital data acquisition.
00003
00004  Copyright (C) 2011-2019 Swabian Instruments
00005  All Rights Reserved
00006
00007  Unauthorized copying of this file is strictly prohibited.
00008  */
00009
00010  #ifndef TT_ITERATORS_H_
00011  #define TT_ITERATORS_H_
00012
00013  #include <algorithm>
00014  #include <array>
00015  #include <assert.h>
00016  #include <atomic>
00017  #include <complex>
00018  #include <deque>
00019  #include <fstream>
00020  #include <functional>
00021  #include <iostream>
00022  #include <limits>
00023  #include <list>
00024  #include <map>
00025  #include <memory>
00026  #include <mutex>
00027  #include <queue>
00028  #include <random>
00029  #include <set>
00030  #include <stdint.h>
00031  #include <stdio.h>
00032  #include <unordered_map>
00033  #include <vector>
00034
00035  // Include mulh helpers on MSVC
00036  #if !defined(__SIZEOF_INT128__) && (defined(_M_X64) || defined(_M_ARM64))
00037  #include <intrin.h>
00038  #endif
00039
00040  #include "TimeTagger.h"
00041
00042  class TT_API FastBinning {
00043  public:
00044      enum class Mode {
00045          ConstZero,
00046          Dividend,
00047          PowerOfTwo,
00048          FixedPoint_32,
00049          FixedPoint_64,
00050          Divide_32,
00051          Divide_64,
00052      };
00053
00054      FastBinning() {}
00055
00056      FastBinning(uint64_t divisor, uint64_t max_duration_);
00057
00058      template <Mode mode> uint64_t divide(uint64_t duration) const {
00059          assert(duration <= max_duration_);
00060          assert(mode == this->mode);
00061          uint64_t out;
00062          switch (mode) {
00063              case Mode::ConstZero:
00064                  out = 0;
00065                  break;
00066              case Mode::Dividend:
00067                  out = duration;
00068                  break;
00069          }
00070      }
00071  };

```

```

00074     case Mode::PowerOfTwo:
00075         out = duration » bits_shift;
00076         break;
00077     case Mode::FixedPoint_32:
00078         out = (duration * factor) » 32;
00079         break;
00080     case Mode::FixedPoint_64:
00081         out = MulHigh(duration, factor);
00082         break;
00083     case Mode::Divide_32:
00084         out = uint32_t(duration) / uint32_t(divisor);
00085         break;
00086     case Mode::Divide_64:
00087         out = duration / divisor;
00088         break;
00089     }
00090     assert(out == duration / divisor);
00091     return out;
00092 }
00093
00094 Mode getMode() const { return mode; }
00095
00096 private:
00097     // returns (a*b) » 64 in a generic but accelerated way
00098     uint64_t MulHigh(uint64_t a, uint64_t b) const {
00099         #ifdef __SIZEOF_INT128__
00100             return ((unsigned __int128)a * (unsigned __int128)b) » 64; // GCC, clang, ...
00101         #elif defined(_M_X64) || defined(_M_ARM64)
00102             return __umulh(a, b); // MSVC
00103         #else
00104             // Generic fallback
00105             uint64_t a_lo = uint32_t(a);
00106             uint64_t a_hi = a » 32;
00107             uint64_t b_lo = uint32_t(b);
00108             uint64_t b_hi = b » 32;
00109
00110             uint64_t a_x_b_hi = a_hi * b_hi;
00111             uint64_t a_x_b_mid = a_hi * b_lo;
00112             uint64_t b_x_a_mid = b_hi * a_lo;
00113             uint64_t a_x_b_lo = a_lo * b_lo;
00114
00115             uint64_t carry_bit = ((uint64_t)(uint32_t)a_x_b_mid + (uint64_t)(uint32_t)b_x_a_mid + (a_x_b_lo »
00116 32)) » 32;
00117
00118             uint64_t multhi = a_x_b_hi + (a_x_b_mid » 32) + (b_x_a_mid » 32) + carry_bit;
00119
00120             return multhi;
00121         #endif
00122     }
00123
00124     uint64_t divisor;
00125     uint64_t max_duration;
00126     uint64_t factor;
00127     int bits_shift;
00128     Mode mode;
00129 };
00130
00131 #define BINNING_TEMPLATE_HELPER(fun_name, binner, ...)
00132 \
00133 \
00134 \
00135 \
00136 \
00137 \
00138 \
00139 \
00140 \
00141 \
00142 \
00143 \
00144 \
00145 \

```

```

00146     fun_name<FastBinning::Mode::FixedPoint_64>(__VA_ARGS__);
00147     \
00148     case FastBinning::Mode::Divide_32:
00149     \
00150         fun_name<FastBinning::Mode::Divide_32>(__VA_ARGS__);
00151     \
00152         break;
00153     \
00154     case FastBinning::Mode::Divide_64:
00155     \
00156         fun_name<FastBinning::Mode::Divide_64>(__VA_ARGS__);
00157     \
00158         break;
00159     \
00160     }
00161 }
00162
00163 class CombinerImpl;
00164 class TT_API Combiner : public IteratorBase {
00165 public:
00166     Combiner(TimeTaggerBase *tagger, std::vector<channel_t> channels);
00167     ~Combiner();
00168     GET_DATA_1D(getChannelCounts, int64_t, array_out, );
00169     GET_DATA_1D(getData, int64_t, array_out, );
00170     channel_t getChannel();
00171 protected:
00172     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00173     override;
00174     void clear_impl() override;
00175 private:
00176     friend class CombinerImpl;
00177     std::unique_ptr<CombinerImpl> impl;
00178 };
00179
00180 class CountBetweenMarkersImpl;
00181 class TT_API CountBetweenMarkers : public IteratorBase {
00182 public:
00183     CountBetweenMarkers(TimeTaggerBase *tagger, channel_t click_channel, channel_t begin_channel,
00184                         channel_t end_channel = CHANNEL_UNUSED, int32_t n_values = 1000);
00185     ~CountBetweenMarkers();
00186     bool ready();
00187     GET_DATA_1D(getData, int32_t, array_out, );
00188     GET_DATA_1D(getBinWidths, timestamp_t, array_out, );
00189     GET_DATA_1D(getIndex, timestamp_t, array_out, );
00190 protected:
00191     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00192     override;
00193     void clear_impl() override;
00194 private:
00195     friend class CountBetweenMarkersImpl;
00196     std::unique_ptr<CountBetweenMarkersImpl> impl;
00197 };
00198
00199 class CounterDataState;
00200 class CounterImpl;
00201 class Counter;
00202 class TT_API CounterData {
00203 public:
00204     ~CounterData();
00205     GET_DATA_2D(getData, int32_t, array_out, );
00206     GET_DATA_2D_OP1(getFrequency, double, array_out, timestamp_t, time_scale, 1000000000000, );
00207     GET_DATA_2D(getDataNormalized, double, array_out, );
00208     GET_DATA_1D(getDataTotalCounts, uint64_t, array_out, );
00209     GET_DATA_1D(getIndex, timestamp_t, array_out, );
00210     GET_DATA_1D(getTime, timestamp_t, array_out, );
00211     GET_DATA_1D(getOverflowMask, signed char, array_out, );

```



```

00390
00394     GET_DATA_1D(getChannels, channel_t, array_out, );
00395
00397     const uint32_t size;
00399     const uint32_t dropped_bins;
00401     const bool overflow;
00402
00403 private:
00404     friend class CounterImpl;
00405     friend class Counter;
00406
00407     CounterData(uint32_t size_, uint32_t dropped_bins_, bool overflow_,
std::shared_ptr<CounterDataState> data_);
00408
00409     const std::shared_ptr<CounterDataState> data;
00410 };
00411
00425 class TT_API Counter : public IteratorBase {
00426 public:
00435     Counter(TimeTaggerBase *tagger, std::vector<channel_t> channels, timestamp_t binwidth = 1000000000,
00436             int32_t n_values = 1);
00437
00438     ~Counter();
00439
00447     GET_DATA_2D_OPI(getData, int32_t, array_out, bool, rolling, true, );
00448
00459     GET_DATA_2D_OPI(getDataNormalized, double, array_out, bool, rolling, true, );
00460
00464     GET_DATA_1D(getDataTotalCounts, uint64_t, array_out, );
00465
00469     GET_DATA_1D(getIndex, timestamp_t, array_out, );
00470
00479     CounterData getDataObject(bool remove = false);
00480
00481 protected:
00482     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
00483     void clear_impl() override;
00484     void on_start() override;
00485
00486 private:
00487     friend class CounterImpl;
00488     std::unique_ptr<CounterImpl> impl;
00489 };
00490
00494 enum class CoincidenceTimestamp : uint32_t {
00495     Last = 0,
00496     Average = 1,
00497     First = 2,
00498     ListedFirst = 3,
00499 };
00500
00501 class CoincidencesImpl;
00514 class TT_API Coincidences : public IteratorBase {
00515 public:
00524     Coincidences(TimeTaggerBase *tagger, std::vector<std::vector<channel_t> coincidenceGroups,
00525             timestamp_t coincidenceWindow, CoincidenceTimestamp timestamp =
CoincidenceTimestamp::Last);
00526
00527     ~Coincidences();
00528
00532     std::vector<channel_t> getChannels();
00533
00534     void setCoincidenceWindow(timestamp_t coincidenceWindow);
00535
00536 protected:
00537     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
00538
00539 private:
00540     friend class CoincidencesImpl;
00541     std::unique_ptr<CoincidencesImpl> impl;
00542 };
00543
00560 class TT_API Coincidence : public Coincidences {
00561 public:
00570     Coincidence(TimeTaggerBase *tagger, std::vector<channel_t> channels, timestamp_t coincidenceWindow =
1000,
00571             CoincidenceTimestamp timestamp = CoincidenceTimestamp::Last)
00572         : Coincidences(tagger, {channels}, coincidenceWindow, timestamp) {}
00573
00577     channel_t getChannel() { return getChannels()[0]; }
00578 };
00579
00580 class CountrateImpl;
00594 class TT_API Countrate : public IteratorBase {
00595 public:

```

```

00602     Countrate(TimeTaggerBase *tagger, std::vector<channel_t> channels);
00603
00604     ~Countrate();
00605
00611     GET_DATA_1D(getData, double, array_out, );
00612
00618     GET_DATA_1D(getCountsTotal, int64_t, array_out, );
00619
00620 protected:
00621     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00622         override;
00623     void clear_impl() override;
00624     void on_start() override;
00625 private:
00626     friend class CountrateImpl;
00627     std::unique_ptr<CountrateImpl> impl;
00628 };
00629
00630 class DelayedChannelImpl;
00641 class TT_API DelayedChannel : public IteratorBase {
00642 public:
00650     DelayedChannel(TimeTaggerBase *tagger, channel_t input_channel, timestamp_t delay);
00651
00652 #ifndef SWIG
00662     DelayedChannel(TimeTaggerBase *tagger, std::vector<channel_t> input_channels, timestamp_t delay);
00663 #endif
00664
00665     ~DelayedChannel();
00666
00673     channel_t getChannel();
00674
00675 #ifndef SWIG
00682     std::vector<channel_t> getChannels();
00683 #endif
00684
00693     void setDelay(timestamp_t delay);
00694
00695 protected:
00696     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00697         override;
00698     void on_start() override;
00699 private:
00700     friend class DelayedChannelImpl;
00701     std::unique_ptr<DelayedChannelImpl> impl;
00702 };
00703
00704 class TriggerOnCountrateImpl;
00729 class TT_API TriggerOnCountrate : public IteratorBase {
00730 public:
00745     TriggerOnCountrate(TimeTaggerBase *tagger, channel_t input_channel, double reference_countrate,
00746         double hysteresis,
00747         timestamp_t time_window);
00748
00749     ~TriggerOnCountrate();
00750
00753     channel_t getChannelAbove();
00754
00758     channel_t getChannelBelow();
00759
00763     std::vector<channel_t> getChannels();
00764
00768     bool isAbove();
00769
00773     bool isBelow();
00774
00778     double getCurrentCountrate();
00779
00789     bool injectCurrentState();
00790
00791 protected:
00792     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00793         override;
00794     void on_start() override;
00795     void clear_impl() override;
00796 private:
00797     friend class TriggerOnCountrateImpl;
00798     std::unique_ptr<TriggerOnCountrateImpl> impl;
00799 };
00800
00804 enum class GatedChannelInitial : uint32_t {
00805     Closed = 0,
00806     Open = 1,
00807 };
00808

```

```

00809 class GatedChannelImpl;
00821 class TT_API GatedChannel : public IteratorBase {
00822 public:
00833     GatedChannel(TimeTaggerBase *tagger, channel_t input_channel, channel_t gate_start_channel,
00834                 channel_t gate_stop_channel, GatedChannelInitial initial =
00835                 GatedChannelInitial::Closed);
00836     ~GatedChannel();
00837
00845     channel_t getChannel();
00846
00847 protected:
00848     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00849     override;
00849     void clear_impl() override;
00850
00851 private:
00852     friend class GatedChannelImpl;
00853     std::unique_ptr<GatedChannelImpl> impl;
00854 };
00855
00856 class FrequencyMultiplierImpl;
00877 class TT_API FrequencyMultiplier : public IteratorBase {
00878 public:
00886     FrequencyMultiplier(TimeTaggerBase *tagger, channel_t input_channel, int32_t multiplier);
00887
00888     ~FrequencyMultiplier();
00889
00890     channel_t getChannel();
00891     int32_t getMultiplier();
00892
00893 protected:
00894     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00895     override;
00895     void clear_impl() override;
00896
00897 private:
00898     friend class FrequencyMultiplierImpl;
00899     std::unique_ptr<FrequencyMultiplierImpl> impl;
00900 };
00901
00902 class IteratorImpl;
00912 class TT_API Iterator : public IteratorBase {
00913 public:
00920     Iterator(TimeTaggerBase *tagger, channel_t channel);
00921
00922     ~Iterator();
00923
00929     timestamp_t next();
00930
00934     uint64_t size();
00935
00936 protected:
00937     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
00938     override;
00938     void clear_impl() override;
00939
00940 private:
00941     friend class IteratorImpl;
00942     std::unique_ptr<IteratorImpl> impl;
00943 };
00944
00945 class TimeTagStreamImpl;
00946 class FileReaderImpl;
00948 class TT_API TimeTagStreamBuffer {
00949     friend class TimeTagStreamImpl;
00950     friend class FileReaderImpl;
00951
00952 public:
00953     ~TimeTagStreamBuffer();
00954
00955     GET_DATA_1D(getOverflows, unsigned char, array_out, ); // deprecated, please use getEventTypes
00956     instead
00956     GET_DATA_1D(getChannels, channel_t, array_out, );
00957     GET_DATA_1D(getTimestamps, timestamp_t, array_out, );
00958     GET_DATA_1D(getMissedEvents, unsigned short, array_out, );
00959     GET_DATA_1D(getEventTypes, unsigned char, array_out, );
00960
00961     uint64_t size;
00962     bool hasOverflows;
00963     timestamp_t tStart;
00964     timestamp_t tGetData;
00965
00966 private:
00967     TimeTagStreamBuffer();
00968
00969     std::vector<channel_t> tagChannels;

```

```

00970     std::vector<timestamp_t> tagTimestamps;
00971     std::vector<unsigned short> tagMissedEvents;
00972     std::vector<Tag::Type> tagTypes;
00973 };
00974
00979 class TT_API TimeTagStream : public IteratorBase {
00980 public:
00990     TimeTagStream(TimeTaggerBase *tagger, uint64_t n_max_events, std::vector<channel_t> channels);
00991     ~TimeTagStream();
00992
00996     uint64_t getCounts();
00997
01001     TimeTagStreamBuffer getData();
01002
01003 protected:
01004     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01005         override;
01006     void clear_impl() override;
01007 private:
01008     friend class TimeTagStreamImpl;
01009     std::unique_ptr<TimeTagStreamImpl> impl;
01010 };
01011
01012 class DumpImpl;
01019 class TT_API Dump : public IteratorBase {
01020 public:
01030     Dump(TimeTaggerBase *tagger, std::string filename, int64_t max_tags,
01031         std::vector<channel_t> channels = std::vector<channel_t>());
01032     ~Dump();
01033
01034 protected:
01035     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01036         override;
01037     void clear_impl() override;
01038     void on_start() override;
01039     void on_stop() override;
01040 private:
01041     friend class DumpImpl;
01042     std::unique_ptr<DumpImpl> impl;
01043 };
01044
01045 class StartStopImpl;
01065 class TT_API StartStop : public IteratorBase {
01066 public:
01075     StartStop(TimeTaggerBase *tagger, channel_t click_channel, channel_t start_channel = CHANNEL_UNUSED,
01076         timestamp_t binwidth = 1000);
01077
01078     ~StartStop();
01079
01080     GET_DATA_2D(getData, timestamp_t, array_out, );
01081
01082 protected:
01083     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01084         override;
01085     void clear_impl() override;
01086     void on_start() override;
01087 private:
01088     friend class StartStopImpl;
01089     std::unique_ptr<StartStopImpl> impl;
01090 };
01091
01092 template <typename T> class TimeDifferencesImpl;
01131 class TT_API TimeDifferences : public IteratorBase {
01132 public:
01144     TimeDifferences(TimeTaggerBase *tagger, channel_t click_channel, channel_t start_channel =
01145         CHANNEL_UNUSED,
01146         channel_t next_channel = CHANNEL_UNUSED, channel_t sync_channel = CHANNEL_UNUSED,
01147         timestamp_t binwidth = 1000, int32_t n_bins = 1000, int32_t n_histograms = 1);
01148
01149     ~TimeDifferences();
01153     GET_DATA_2D(getData, int32_t, array_out, );
01154
01158     GET_DATA_1D(getIndex, timestamp_t, array_out, );
01159
01165     void setMaxCounts(uint64_t max_counts);
01166
01170     uint64_t getCounts();
01171
01180     int32_t getHistogramIndex() const;
01181
01185     bool ready();
01186
01187 protected:

```

```

01188     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01189     override;
01189     void clear_impl() override;
01190     void on_start() override;
01191
01192 private:
01193     friend class TimeDifferencesImpl<TimeDifferences>;
01194     std::unique_ptr<TimeDifferencesImpl<TimeDifferences>» impl;
01195 };
01196
01197 template <typename T> class HistogramNDImpl;
01211 class TT_API Histogram2D : public IteratorBase {
01212 public:
01225     Histogram2D(TimeTaggerBase *tagger, channel_t start_channel, channel_t stop_channel_1, channel_t
01226 stop_channel_2,
01226         timestamp_t binwidth_1, timestamp_t binwidth_2, int32_t n_bins_1, int32_t n_bins_2);
01227     ~Histogram2D();
01228
01232     GET_DATA_2D(getData, int32_t, array_out, );
01233
01238     GET_DATA_3D(getIndex, timestamp_t, array_out, );
01239
01243     GET_DATA_1D(getIndex_1, timestamp_t, array_out, );
01244
01248     GET_DATA_1D(getIndex_2, timestamp_t, array_out, );
01249
01250 protected:
01251     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01252     override;
01252     void clear_impl() override;
01253
01254 private:
01255     friend class HistogramNDImpl<Histogram2D>;
01256     std::unique_ptr<HistogramNDImpl<Histogram2D>» impl;
01257 };
01258
01269 class TT_API HistogramND : public IteratorBase {
01270 public:
01279     HistogramND(TimeTaggerBase *tagger, channel_t start_channel, std::vector<channel_t> stop_channels,
01280         std::vector<timestamp_t> binwidths, std::vector<int32_t> n_bins);
01281     ~HistogramND();
01282
01288     GET_DATA_1D(getData, int32_t, array_out, );
01289
01293     GET_DATA_1D_Op1(getIndex, timestamp_t, array_out, int32_t, dim, 0, );
01294
01295 protected:
01296     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01297     override;
01297     void clear_impl() override;
01298
01299 private:
01300     friend class HistogramNDImpl<HistogramND>;
01301     std::unique_ptr<HistogramNDImpl<HistogramND>» impl;
01302 };
01303
01304 class TimeDifferencesNDImpl;
01333 class TT_API TimeDifferencesND : public IteratorBase {
01334 public:
01349     TimeDifferencesND(TimeTaggerBase *tagger, channel_t click_channel, channel_t start_channel,
01350         std::vector<channel_t> next_channels, std::vector<channel_t> sync_channels,
01351         std::vector<int32_t> n_histograms, timestamp_t binwidth, int32_t n_bins);
01352
01353     ~TimeDifferencesND();
01354
01358     GET_DATA_2D(getData, int32_t, array_out, );
01359
01363     GET_DATA_1D(getIndex, timestamp_t, array_out, );
01364
01365 protected:
01366     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01367     override;
01367     void clear_impl() override;
01368     void on_start() override;
01369
01370 private:
01371     friend class TimeDifferencesNDImpl;
01372     std::unique_ptr<TimeDifferencesNDImpl> impl;
01373 };
01374
01394 class TT_API Histogram : public IteratorBase {
01395 public:
01406     Histogram(TimeTaggerBase *tagger, channel_t click_channel, channel_t start_channel = CHANNEL_UNUSED,
01407         timestamp_t binwidth = 1000, int32_t n_bins = 1000);
01408
01409     ~Histogram();
01410

```

```

01411     GET_DATA_1D(getData, int32_t, array_out, );
01412
01413     GET_DATA_1D(getIndex, timestamp_t, array_out, );
01414
01415 protected:
01416     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01417         override;
01418     void clear_impl() override;
01419     void on_start() override;
01420 private:
01421     friend class TimeDifferencesImpl<Histogram>;
01422     std::unique_ptr<TimeDifferencesImpl<Histogram> > impl;
01423 };
01424
01425 struct FrequencyCounterDataImpl;
01426
01427 class TT_API FrequencyCounterData {
01428 public:
01429     ~FrequencyCounterData();
01430
01431     GET_DATA_1D(getIndex, timestamp_t, array_out, );
01432
01433     GET_DATA_1D(getTime, timestamp_t, array_out, );
01434
01435     GET_DATA_2D(getOverflowMask, signed char, array_out, );
01436
01437     GET_DATA_2D(getPeriodsCount, timestamp_t, array_out, );
01438
01439     GET_DATA_2D(getPeriodsFraction, double, array_out, );
01440
01441     GET_DATA_2D_Op1(getFrequency, double, array_out, timestamp_t, time_scale, 1000000000000, );
01442
01443     GET_DATA_2D(getFrequencyInstantaneous, double, array_out, );
01444
01445     GET_DATA_2D_Op1(getPhase, double, array_out, double, reference_frequency, 0, );
01446
01447     const timestamp_t overflow_samples;
01448
01449     const unsigned int size;
01450
01451     const bool align_to_reference;
01452
01453     const timestamp_t sampling_interval;
01454
01455     const timestamp_t sample_offset;
01456
01457     const bool channels_last_dim;
01458 private:
01459     FrequencyCounterData(timestamp_t overflow_samples, unsigned int size, bool align_to_reference,
01460         timestamp_t sampling_interval, timestamp_t index_offset, bool
01461         channels_last_dim);
01462     friend class FrequencyCounter;
01463
01464     std::shared_ptr<FrequencyCounterDataImpl> data;
01465 };
01466
01467 class FrequencyCounterImpl;
01468
01469 class TT_API FrequencyCounter : public IteratorBase {
01470 public:
01471     FrequencyCounter(TimeTaggerBase *tagger, std::vector<channel_t> channels, timestamp_t
01472         sampling_interval,
01473         timestamp_t fitting_window, int32_t n_values = 0);
01474     ~FrequencyCounter();
01475     FrequencyCounterData getDataObject(uint16_t event_divider = 1, bool remove = false, bool
01476         channels_last_dim = false);
01477
01478 protected:
01479     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01480         override;
01481     void clear_impl() override;
01482     void on_start() override;
01483 private:
01484     friend class FrequencyCounterImpl;
01485     std::unique_ptr<FrequencyCounterImpl> impl;
01486 };
01487
01488 class HistogramLogBinsImpl;
01489
01490 struct HistogramLogBinsDataImpl;
01491
01492 class TT_API HistogramLogBinsData {
01493 public:
01494     ~HistogramLogBinsData();

```

```

01539
01543     GET_DATA_1D(getCounts, uint64_t, array_out, );
01544
01548     GET_DATA_1D(getG2Normalization, double, array_out, );
01549
01553     GET_DATA_1D(getG2, double, array_out, );
01554
01555     const timestamp_t accumulation_time_start;
01556     const timestamp_t accumulation_time_click;
01557
01558 private:
01559     HistogramLogBinsData(timestamp_t accumulation_time_start, timestamp_t accumulation_time_click);
01560     friend HistogramLogBinsImpl;
01561
01562     std::shared_ptr<HistogramLogBinsDataImpl> data;
01563 };
01564
01565 struct TT_API ChannelGate {
01566     ChannelGate(channel_t gate_open_channel, channel_t gate_close_channel,
01567                 GatedChannelInitial initial = GatedChannelInitial::Open)
01568         : gate_open_channel(gate_open_channel), gate_close_channel(gate_close_channel),
01569           initial(initial){};
01569     const channel_t gate_open_channel;
01570     const channel_t gate_close_channel;
01571     const GatedChannelInitial initial;
01572 };
01573
01588 class TT_API HistogramLogBins : public IteratorBase {
01589 public:
01604     HistogramLogBins(TimeTaggerBase *tagger, channel_t click_channel, channel_t start_channel, double
01605 exp_start,
01606                     double exp_stop, int32_t n_bins, const ChannelGate *click_gate = nullptr,
01607                     const ChannelGate *start_gate = nullptr);
01607     ~HistogramLogBins();
01608
01609     HistogramLogBinsData getDataObject();
01610
01614     GET_DATA_1D(getData, uint64_t, array_out, );
01615
01619     GET_DATA_1D(getDataNormalizedCountsPerPs, double, array_out, );
01620
01626     GET_DATA_1D(getDataNormalizedG2, double, array_out, );
01627
01631     GET_DATA_1D(getBinEdges, timestamp_t, array_out, );
01632
01633 protected:
01634     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01635         override;
01635     void clear_impl() override;
01636
01637 private:
01638     friend class HistogramLogBinsImpl;
01639     std::unique_ptr<HistogramLogBinsImpl> impl;
01640 };
01641
01642 class CorrelationImpl;
01655 class TT_API Correlation : public IteratorBase {
01656 public:
01669     Correlation(TimeTaggerBase *tagger, channel_t channel_1, channel_t channel_2 = CHANNEL_UNUSED,
01670                 timestamp_t binwidth = 1000, int n_bins = 1000);
01671
01673     ~Correlation();
01674
01680     GET_DATA_1D(getData, int32_t, array_out, );
01681
01693     GET_DATA_1D(getDataNormalized, double, array_out, );
01694
01700     GET_DATA_1D(getIndex, timestamp_t, array_out, );
01701
01702 protected:
01703     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01704         override;
01704     void clear_impl() override;
01705
01706 private:
01707     friend class CorrelationImpl;
01708     std::unique_ptr<CorrelationImpl> impl;
01709 };
01710
01712 enum State {
01713     UNKNOWN,
01714     HIGH,
01715     LOW,
01716 };
01718 struct Event {
01719     timestamp_t time;
01720     State state;

```

```

01721 };
01722 class ScopeImpl;
01737 class TT_API Scope : public IteratorBase {
01738 public:
01749     Scope(TimeTaggerBase *tagger, std::vector<channel_t> event_channels, channel_t trigger_channel,
01750           timestamp_t window_size = 1000000000, int32_t n_traces = 1, int32_t n_max_events = 1000);
01751
01752     ~Scope();
01753
01754     bool ready();
01755
01756     int32_t triggered();
01757
01758     std::vector<std::vector<Event>> getData();
01759
01760     timestamp_t getWindowSize();
01761
01762 protected:
01763     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01764         override;
01765     void clear_impl() override;
01766 private:
01767     friend class ScopeImpl;
01768     std::unique_ptr<ScopeImpl> impl;
01769 };
01770
01771 class TimeTaggerProxy;
01784 class TT_API SynchronizedMeasurements {
01785 public:
01791     SynchronizedMeasurements(TimeTaggerBase *tagger);
01792
01793     ~SynchronizedMeasurements();
01794
01801     void registerMeasurement(IteratorBase *measurement);
01802
01809     void unregisterMeasurement(IteratorBase *measurement);
01810
01814     void clear();
01815
01819     void start();
01820
01824     void stop();
01825
01829     void startFor(timestamp_t capture_duration, bool clear = true);
01830
01840     bool waitUntilFinished(int64_t timeout = -1);
01841
01845     bool isRunning();
01846
01852     TimeTaggerBase *getTagger();
01853
01854 protected:
01861     void runCallback(TimeTaggerBase::IteratorCallback callback, std::unique_lock<std::mutex> &lk, bool
01862         block = true);
01863 private:
01864     friend class TimeTaggerProxy;
01865
01866     void release();
01867
01868     std::set<IteratorBase *> registered_measurements;
01869     std::mutex measurements_mutex;
01870     TimeTaggerBase *tagger;
01871     bool has_been_released = false;
01872     std::unique_ptr<TimeTaggerProxy> proxy;
01873 };
01874
01875 class ConstantFractionDiscriminatorImpl;
01885 class TT_API ConstantFractionDiscriminator : public IteratorBase {
01886 public:
01894     ConstantFractionDiscriminator(TimeTaggerBase *tagger, std::vector<channel_t> channels, timestamp_t
01895         search_window);
01896
01897     ~ConstantFractionDiscriminator();
01898
01904     std::vector<channel_t> getChannels();
01905
01906 protected:
01907     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01908         override;
01909     void on_start() override;
01910 private:
01911     friend class ConstantFractionDiscriminatorImpl;
01912     std::unique_ptr<ConstantFractionDiscriminatorImpl> impl;
01913 };

```



```

01914
01915 class FileWriterImpl;
01920 class TT_API FileWriter : public IteratorBase {
01921 public:
01929     FileWriter(TimeTaggerBase *tagger, const std::string &filename, std::vector<channel_t> channels);
01930     ~FileWriter();
01931
01937     void split(const std::string &new_filename = "");
01938
01946     void setMaxFileSize(uint64_t max_file_size);
01947
01953     uint64_t getMaxFileSize();
01954
01960     uint64_t getTotalEvents();
01961
01967     uint64_t getTotalSize();
01968
01974     void setMarker(const std::string &marker);
01975
01976 protected:
01977     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
01978         override;
01978     void clear_impl() override;
01979     void on_start() override;
01980     void on_stop() override;
01981
01982 private:
01983     friend class FileWriterImpl;
01984     std::unique_ptr<FileWriterImpl> impl;
01985 };
01986
01994 class TT_API FileReader {
01995 public:
02004     FileReader(std::vector<std::string> filenames);
02005
02013     FileReader(const std::string &filename);
02014     ~FileReader();
02015
02021     bool hasData();
02022
02031     TimeTagStreamBuffer getData(uint64_t n_events);
02032
02041     bool getDataRaw(std::vector<Tag> &tag_buffer);
02042
02049     std::string getConfiguration();
02050
02056     std::vector<channel_t> getChannelList();
02057
02063     std::string getLastMarker();
02064
02065 private:
02066     friend class FileReaderImpl;
02067     std::unique_ptr<FileReaderImpl> impl;
02068 };
02069
02070 class EventGeneratorImpl;
02085 class TT_API EventGenerator : public IteratorBase {
02086 public:
02097     EventGenerator(TimeTaggerBase *tagger, channel_t trigger_channel, std::vector<timestamp_t> pattern,
02098         uint64_t trigger_divider = 1, uint64_t divider_offset = 0, channel_t stop_channel =
02099         CHANNEL_UNUSED);
02100     ~EventGenerator();
02101
02109     channel_t getChannel();
02110
02111 protected:
02112     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
02113         override;
02113     void clear_impl() override;
02114     void on_start() override;
02115
02116 private:
02117     friend class EventGeneratorImpl;
02118     std::unique_ptr<EventGeneratorImpl> impl;
02119 };
02120
02121 class CombinationsImpl;
02142 class TT_API Combinations : public IteratorBase {
02143 public:
02151     Combinations(TimeTaggerBase *tagger, std::vector<channel_t> const &channels, timestamp_t
02152         window_size);
02153     ~Combinations();
02154
02159     channel_t getChannel(std::vector<channel_t> const &input_channels) const;
02160

```

```

02162     channel_t getSumChannel(int n_channels) const;
02163
02164     std::vector<channel_t> getCombination(channel_t virtual_channel) const;
02165
02166 protected:
02167     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
02168     override;
02169     void clear_impl() override;
02170
02171 private:
02172     friend class CombinationsImpl;
02173     std::unique_ptr<CombinationsImpl> impl;
02174 };
02175
02181 class TT_API CustomMeasurementBase : public IteratorBase {
02182 protected:
02183     // Only usable for subclasses.
02184     CustomMeasurementBase(TimeTaggerBase *tagger);
02185
02186 public:
02187     ~CustomMeasurementBase() override;
02188
02189     // Stop all running custom measurements. Use this to avoid races on shutdown the target language.
02190     static void stop_all_custom_measurements();
02191
02192     // Forward the public API of the measurement
02193     void register_channel(channel_t channel);
02194     void unregister_channel(channel_t channel);
02195     void finalize_init();
02196     bool is_running() const;
02197     void _lock();
02198     void _unlock();
02199
02200 protected:
02201     // By default, this calls next_impl_cs
02202     virtual bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t
02203     end_time) override;
02204
02205     // Handler with easier to wrap API. By default, this does nothing
02206     virtual void next_impl_cs(void *tags_ptr, uint64_t num_tags, timestamp_t begin_time, timestamp_t
02207     end_time);
02208
02209     // Forward the public handlers for swig to detect this virtual methods. By default, they do nothing
02210     virtual void clear_impl() override;
02211     virtual void on_start() override;
02212     virtual void on_stop() override;
02213 };
02214
02215 class TT_API FlimAbstract : public IteratorBase {
02216 public:
02217     FlimAbstract(TimeTaggerBase *tagger, channel_t start_channel, channel_t click_channel, channel_t
02218     pixel_begin_channel,
02219     uint32_t n_pixels, uint32_t n_bins, timestamp_t binwidth, channel_t pixel_end_channel =
02220     CHANNEL_UNUSED,
02221     channel_t frame_begin_channel = CHANNEL_UNUSED, uint32_t finish_after_outputframe = 0,
02222     uint32_t n_frame_average = 1, bool pre_initialize = true);
02223
02224     ~FlimAbstract();
02225
02226     bool isAcquiring() const { return acquiring; }
02227
02228 protected:
02229     template <FastBinning::Mode bin_mode> void process_tags(const std::vector<Tag> &incoming_tags);
02230     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
02231     override;
02232     void clear_impl() override;
02233     void on_start() override;
02234
02235     virtual void on_frame_end() = 0;
02236
02237     const channel_t start_channel;
02238     const channel_t click_channel;
02239     const channel_t pixel_begin_channel;
02240     const uint32_t n_pixels;
02241     const uint32_t n_bins;
02242     const timestamp_t binwidth;
02243     const channel_t pixel_end_channel;
02244     const channel_t frame_begin_channel;
02245     const uint32_t finish_after_outputframe;
02246     const uint32_t n_frame_average;
02247
02248     const timestamp_t time_window;
02249
02250     timestamp_t current_frame_begin;
02251     timestamp_t current_frame_end;
02252
02253     // state

```

```

02277     bool acquiring{};
02278     bool frame_acquisition{};
02279     bool pixel_acquisition{};
02280
02281     uint32_t pixels_processed{};
02282     uint32_t frames_completed{};
02283     uint32_t ticks{};
02284     size_t data_base{};
02285
02286     std::vector<uint32_t> frame;
02287
02288     std::vector<timestamp_t> pixel_begins;
02289     std::vector<timestamp_t> pixel_ends;
02290     std::deque<timestamp_t> previous_starts;
02291
02292     FastBinning binner;
02293
02294     std::recursive_mutex acquisition_lock;
02295     bool initialized;
02296 };
02297
02307 class TT_API FlimBase : public FlimAbstract {
02308 public:
02328     FlimBase(TimeTaggerBase *tagger, channel_t start_channel, channel_t click_channel, channel_t
pixel_begin_channel,
02329             uint32_t n_pixels, uint32_t n_bins, timestamp_t binwidth, channel_t pixel_end_channel =
CHANNEL_UNUSED,
02330             channel_t frame_begin_channel = CHANNEL_UNUSED, uint32_t finish_after_outputframe = 0,
uint32_t n_frame_average = 1, bool pre_initialize = true);
02331
02332
02333     ~FlimBase();
02334
02342     void initialize();
02343
02344 protected:
02345     void on_frame_end() override final;
02346
02347     virtual void frameReady(uint32_t frame_number, std::vector<uint32_t> &data,
std::vector<timestamp_t> &pixel_begin_times, std::vector<timestamp_t>
02348 &pixel_end_times,
                                timestamp_t frame_begin_time, timestamp_t frame_end_time);
02349
02350
02351     uint32_t total_frames;
02352 };
02353
02355 class TT_API FlimFrameInfo {
02356     friend class Flim;
02357
02358 public:
02359     ~FlimFrameInfo();
02360
02370     int32_t getFrameNumber() const { return frame_number; }
02371
02382     bool isValid() const { return valid; }
02383
02391     uint32_t getPixelPosition() const { return pixel_position; }
02392
02393     GET_DATA_2D(getHistograms, uint32_t, array_out, );
02394     GET_DATA_1D(getIntensities, float, array_out, );
02395     GET_DATA_1D(getSummedCounts, uint64_t, array_out, );
02396     GET_DATA_1D(getPixelBegins, timestamp_t, array_out, );
02397     GET_DATA_1D(getPixelEnds, timestamp_t, array_out, );
02398
02399 private:
02400     FlimFrameInfo();
02401     std::vector<uint32_t> histograms;
02402     std::vector<timestamp_t> pixel_begins;
02403     std::vector<timestamp_t> pixel_ends;
02404
02405 public:
02406     uint32_t pixels;
02407     uint32_t bins;
02408     int32_t frame_number;
02409     uint32_t pixel_position;
02410     bool valid;
02411 };
02412
02433 class TT_API Flim : public FlimAbstract {
02434 public:
02454     Flim(TimeTaggerBase *tagger, channel_t start_channel, channel_t click_channel, channel_t
pixel_begin_channel,
02455         uint32_t n_pixels, uint32_t n_bins, timestamp_t binwidth, channel_t pixel_end_channel =
CHANNEL_UNUSED,
02456         channel_t frame_begin_channel = CHANNEL_UNUSED, uint32_t finish_after_outputframe = 0,
uint32_t n_frame_average = 1, bool pre_initialize = true);
02457
02458
02459     ~Flim();

```

```

02460
02468     void initialize();
02469
02481     GET_DATA_2D_OP1(getReadyFrame, uint32_t, array_out, int32_t, index, -1, );
02482
02497     GET_DATA_1D_OP1(getReadyFrameIntensity, float, array_out, int32_t, index, -1, );
02498
02504     GET_DATA_2D(getCurrentFrame, uint32_t, array_out, );
02505
02514     GET_DATA_1D(getCurrentFrameIntensity, float, array_out, );
02515
02527     GET_DATA_2D_OP2(getSummedFrames, uint32_t, array_out, bool, only_ready_frames, true, bool,
clear_summed, false, );
02528
02541     GET_DATA_1D_OP2(getSummedFramesIntensity, float, array_out, bool, only_ready_frames, true, bool,
clear_summed,
false, );
02542
02543
02554     FlimFrameInfo getReadyFrameEx(int32_t index = -1);
02555
02561     FlimFrameInfo getCurrentFrameEx();
02562
02572     FlimFrameInfo getSummedFramesEx(bool only_ready_frames = true, bool clear_summed = false);
02573
02580     uint32_t getFramesAcquired() const { return total_frames; }
02581
02587     GET_DATA_1D(getIndex, timestamp_t, array_out, );
02588
02589 protected:
02590     void on_frame_end() override final;
02591     void clear_impl() override;
02592
02593     uint32_t get_ready_index(int32_t index);
02594
02595     virtual void frameReady(uint32_t frame_number, std::vector<uint32_t> &data,
std::vector<timestamp_t> &pixel_begin_times, std::vector<timestamp_t>
&pixel_end_times,
timestamp_t frame_begin_time, timestamp_t frame_end_time);
02598
02599     std::vector<std::vector<uint32_t>> back_frames;
02600     std::vector<std::vector<timestamp_t>> frame_begins;
02601     std::vector<std::vector<timestamp_t>> frame_ends;
02602     std::vector<uint32_t> pixels_completed;
02603     std::vector<uint32_t> summed_frames;
02604     std::vector<timestamp_t> accum_diffs;
02605     uint32_t captured_frames;
02606     uint32_t total_frames;
02607     int32_t last_frame;
02608
02609     std::mutex swap_chain_lock;
02610 };
02611
02612 class SamplerImpl;
02627 class TT_API Sampler : public IteratorBase {
02628 public:
02637     Sampler(TimeTaggerBase *tagger, channel_t trigger, std::vector<channel_t> channels, size_t
max_triggers);
02638     ~Sampler();
02639
02654     GET_DATA_2D(getData, timestamp_t, array_out, );
02655
02669     GET_DATA_2D(getDataAsMask, timestamp_t, array_out, );
02670
02671 protected:
02672     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
02673     void clear_impl() override;
02674     void on_start() override;
02675
02676 private:
02677     friend class SamplerImpl;
02678     std::unique_ptr<SamplerImpl> impl;
02679 };
02680
02681 class SyntheticSingleTagImpl;
02693 class TT_API SyntheticSingleTag : public IteratorBase {
02694 public:
02702     SyntheticSingleTag(TimeTaggerBase *tagger, channel_t base_channel = CHANNEL_UNUSED);
02703     ~SyntheticSingleTag();
02704
02708     void trigger();
02709
02710     channel_t getChannel() const;
02711
02712 protected:
02713     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;

```

```

02714
02715 private:
02716     friend class SyntheticSingleTagImpl;
02717     std::unique_ptr<SyntheticSingleTagImpl> impl;
02718 };
02719
02720 class FrequencyStabilityImpl;
02721 struct FrequencyStabilityDataImpl;
02722 class FrequencyStability;
02723
02724 class TT_API FrequencyStabilityData {
02725 public:
02726     ~FrequencyStabilityData();
02727
02728     GET_DATA_1D(getSTDD, double, array_out, );
02729
02730     GET_DATA_1D(getADEV, double, array_out, );
02731
02732     GET_DATA_1D(getMDEV, double, array_out, );
02733
02734     GET_DATA_1D(getTDEV, double, array_out, );
02735
02736     GET_DATA_1D(getHDEV, double, array_out, );
02737
02738     GET_DATA_1D(getADEVScaled, double, array_out, );
02739
02740     GET_DATA_1D(getHDEVScaled, double, array_out, );
02741
02742     GET_DATA_1D(getTau, double, array_out, );
02743
02744     GET_DATA_1D(getTracePhase, double, array_out, );
02745
02746     GET_DATA_1D(getTraceFrequency, double, array_out, );
02747
02748     GET_DATA_1D_OPL(getTraceFrequencyAbsolute, double, array_out, double, input_frequency, 0.0, );
02749
02750     GET_DATA_1D(getTraceIndex, double, array_out, );
02751
02752 private:
02753     FrequencyStabilityData();
02754     friend class FrequencyStability;
02755
02756     std::shared_ptr<FrequencyStabilityDataImpl> data;
02757 };
02758
02759 class TT_API FrequencyStability : public IteratorBase {
02760 public:
02761     FrequencyStability(TimeTaggerBase *tagger, channel_t channel, std::vector<uint64_t> steps,
02762         timestamp_t average = 1000,
02763         uint64_t trace_len = 1000);
02764     ~FrequencyStability();
02765
02766     FrequencyStabilityData getDataObject();
02767
02768 protected:
02769     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
02770         override;
02771     void clear_impl() override;
02772     void on_start() override;
02773
02774 private:
02775     friend class FrequencyStabilityImpl;
02776     std::unique_ptr<FrequencyStabilityImpl> impl;
02777 };
02778
02779 class PRBS;
02780
02781 namespace Experimental {
02782
02783     class PulsePerSecondImpl;
02784     class PulsePerSecondDataState;
02785
02786     class TT_API PulsePerSecondData {
02787     public:
02788         GET_DATA_1D(getIndices, int64_t, array_out, );
02789         GET_DATA_1D(getReferenceOffsets, double, array_out, );
02790         GET_DATA_2D(getSignalOffsets, double, array_out, );
02791         GET_DATA_1D(getUtcSeconds, double, array_out, );
02792         std::vector<std::string> getUtcDates();
02793         GET_DATA_1D(getStatus, bool, array_out, );
02794         const size_t size;
02795         ~PulsePerSecondData();
02796
02797     private:
02798         PulsePerSecondData(std::shared_ptr<PulsePerSecondDataState> data_ptr, const std::vector<channel_t>
02799             channel_list,
02800             size_t size);

```

```

02927     std::shared_ptr<PulsePerSecondDataState> data;
02928     const std::vector<channel_t> channel_list;
02929     friend PulsePerSecondDataState;
02930     friend PulsePerSecondImpl;
02931 };
02932
02950 class TT_API PulsePerSecondMonitor : public IteratorBase {
02951 public:
02962     PulsePerSecondMonitor(TimeTaggerBase *tagger, channel_t reference_channel, std::vector<channel_t>
signal_channels,
02963                             std::string filename = "", timestamp_t period = 1E12);
02964     ~PulsePerSecondMonitor();
02971     PulsePerSecondData getDataObject(bool remove = false);
02972
02973 protected:
02974     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
02975     void clear_impl() override;
02976     void on_start() override;
02977
02978 private:
02979     friend class PulsePerSecondImpl;
02980     std::unique_ptr<PulsePerSecondImpl> impl;
02981 };
02982
02983 class SignalGeneratorBaseImpl;
02984
02985 class TT_API SignalGeneratorBase : public IteratorBase {
02986 public:
02987     SignalGeneratorBase(TimeTaggerBase *tagger, channel_t base_channel = CHANNEL_UNUSED);
02988     ~SignalGeneratorBase();
02989
02997     channel_t getChannel();
02998
02999     // void registerReactor(std::string property, channel_t trigger_channel, std::vector<float> values,
bool repeat);
03000
03001 protected:
03002     virtual void initialize(timestamp_t initial_time) = 0;
03003     virtual timestamp_t get_next() = 0;
03004
03005     // void addReactable(std::string property, std::function<void, float> &&callback);
03006
03007     virtual void on_restart(timestamp_t restart_time);
03008
03009     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
03010     void on_stop() override;
03011
03012     // callbacks
03013     bool isProcessingFinished();
03014     void set_processing_finished(bool is_finished);
03015
03016     friend class SignalGeneratorBaseImpl;
03017     std::unique_ptr<SignalGeneratorBaseImpl> impl;
03018 };
03019
03020 class TT_API PhotonGenerator : public SignalGeneratorBase {
03021 public:
03033     PhotonGenerator(TimeTaggerBase *tagger, double countrate, channel_t base_channel, int32_t seed =
-1);
03034     ~PhotonGenerator();
03035     void finalize_init();
03036     void set_T_PERIOD(timestamp_t new_T);
03037     timestamp_t get_T_PERIOD();
03038
03039 protected:
03040     void initialize(timestamp_t initial_time) override;
03041     void on_restart(timestamp_t restart_time) override;
03042     timestamp_t get_next() override;
03043     virtual double get_intensity() = 0;
03044     timestamp_t T_PERIOD;
03045
03046 private:
03047     timestamp_t get_new_stamp();
03048     std::minstd_rand0 generator;
03049     std::exponential_distribution<double> exp_distribution;
03050     timestamp_t accumulated;
03051     timestamp_t base_time;
03052     timestamp_t t_evolution;
03053     double current_intensity;
03054 };
03055
03056 class DlsSignalGeneratorImpl;
03057
03058 class TT_API DlsSignalGenerator : public PhotonGenerator {
03059 public:

```

```

03071     DlsSignalGenerator(TimeTaggerBase *tagger, double decay_time, double countrate,
03072                        channel_t output_channel = CHANNEL_UNUSED, int32_t seed = -1);
03073     DlsSignalGenerator(TimeTaggerBase *tagger, std::vector<double> decay_times, double countrate,
03074                        channel_t output_channel = CHANNEL_UNUSED, int32_t seed = -1);
03075     ~DlsSignalGenerator();
03076     unsigned int get_N();
03077
03078 protected:
03079     double get_intensity() override;
03080
03081 private:
03082     friend class DlsSignalGeneratorImpl;
03083     std::unique_ptr<DlsSignalGeneratorImpl> impl;
03084 };
03085
03086 class FcsSignalGeneratorImpl;
03087
03088 class TT_API FcsSignalGenerator : public PhotonGenerator {
03089 public:
03102     FcsSignalGenerator(TimeTaggerBase *tagger, double correlation_time, double N_focus, double
countrate,
03103                        channel_t output_channel = CHANNEL_UNUSED, int32_t seed = -1);
03104     ~FcsSignalGenerator();
03105     unsigned int get_N();
03106     void set_boundary_limit(double new_boundary);
03107
03108 protected:
03109     double get_intensity() override;
03110
03111 private:
03112     friend class FcsSignalGeneratorImpl;
03113     std::unique_ptr<FcsSignalGeneratorImpl> impl;
03114 };
03115
03116 class TT_API UniformSignalGenerator : public SignalGeneratorBase {
03117 public:
03129     UniformSignalGenerator(TimeTaggerBase *tagger, timestamp_t upper_bound, timestamp_t lower_bound = 1,
03130                            channel_t base_channel = CHANNEL_UNUSED, int32_t seed = -1);
03131     ~UniformSignalGenerator();
03132
03133 protected:
03134     void initialize(timestamp_t initial_time) override;
03135     timestamp_t get_next() override;
03136
03137     void on_restart(timestamp_t restart_time) override;
03138
03139 private:
03140     std::unique_ptr<PRBS> generator;
03141     timestamp_t lower_bound;
03142     timestamp_t period;
03143     timestamp_t accumulated;
03144     timestamp_t base_time;
03145 };
03146
03147 class TT_API GaussianSignalGenerator : public SignalGeneratorBase {
03148 public:
03160     GaussianSignalGenerator(TimeTaggerBase *tagger, double mean, double standard_deviation,
03161                             channel_t base_channel = CHANNEL_UNUSED, int32_t seed = -1);
03162     ~GaussianSignalGenerator();
03163
03164 protected:
03165     void initialize(timestamp_t initial_time) override;
03166     timestamp_t get_next() override;
03167
03168     void on_restart(timestamp_t restart_time) override;
03169
03170 private:
03171     std::minstd_rand0 generator;
03172     std::normal_distribution<double> distr;
03173     timestamp_t accumulated;
03174     timestamp_t base_time;
03175 };
03176
03177 class FlickerDistributionVossMcCartney;
03178
03179 class TT_API OscillatorSimulation : public SignalGeneratorBase {
03180 public:
03201     OscillatorSimulation(TimeTaggerBase *tagger, double nominal_frequency, double coeff_phase_white =
0.0,
03202                          double coeff_phase_flicker = 0.0, double coeff_freq_white = 0.0, double
coeff_freq_flicker = 0.0,
03203                          double coeff_random_drift = 0.0, double coeff_linear_drift = 0.0,
03204                          channel_t base_channel = CHANNEL_UNUSED, int32_t seed = -1);
03205
03206     ~OscillatorSimulation();
03207
03208 protected:

```

```

03209 void initialize(timestamp_t initial_time) override;
03210 timestamp_t get_next() override;
03211
03212 void on_restart(timestamp_t restart_time) override;
03213
03214 private:
03215     double const coeff_phase_white, coeff_phase_flicker, coeff_freq_white, coeff_freq_flicker,
03216     coeff_random_drift,
03217     coeff_linear_drift;
03218     timestamp_t const nominal_period_int;
03219     double const nominal_period_fractional;
03220
03221     std::unique_ptr<FlickerDistributionVossMcCartney> flicker_phase, flicker_freq;
03222     std::normal_distribution<double> white;
03223     std::mt19937_64 generator;
03224     double freq_random_walk_acc,
03225     fractional_ps_acc;
03226     timestamp_t last_time;
03227     uint64_t num_periods_passed{};
03228 };
03229
03230 class TT_API TwoStateExponentialSignalGenerator : public SignalGeneratorBase {
03231 public:
03243     TwoStateExponentialSignalGenerator(TimeTaggerBase *tagger, double excitation_time, double life_time,
03244     channel_t base_channel = CHANNEL_UNUSED, int32_t seed = -1);
03245     ~TwoStateExponentialSignalGenerator();
03246
03247 protected:
03248     void initialize(timestamp_t initial_time) override;
03249     timestamp_t get_next() override;
03250
03251     void on_restart(timestamp_t restart_time) override;
03252
03253 private:
03254     std::minstd_rand0 generator;
03255     std::exponential_distribution<double> excitation_time_distr;
03256     std::exponential_distribution<double> life_time_distr;
03257     timestamp_t accumulated;
03258     timestamp_t base_time;
03259 };
03260
03261 class MarkovProcessGeneratorImpl;
03262 class TT_API MarkovProcessGenerator : public IteratorBase {
03263 public:
03278     MarkovProcessGenerator(TimeTaggerBase *tagger, uint64_t num_states, std::vector<double> frequencies,
03279     std::vector<channel_t> ref_channels,
03280     std::vector<channel_t> base_channels = std::vector<channel_t>(), int32_t seed
03281     = -1);
03282     ~MarkovProcessGenerator();
03283     channel_t getChannel();
03284     std::vector<channel_t> getChannels();
03285
03286 protected:
03287     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
03288     override;
03289     void on_stop() override;
03290
03291 private:
03292     friend class MarkovProcessGeneratorImpl;
03293     std::unique_ptr<MarkovProcessGeneratorImpl> impl;
03294 };
03295
03296 class TT_API ExponentialSignalGenerator : public SignalGeneratorBase {
03297 public:
03307     ExponentialSignalGenerator(TimeTaggerBase *tagger, double rate, channel_t base_channel =
03308     CHANNEL_UNUSED,
03309     int32_t seed = -1);
03310     ~ExponentialSignalGenerator();
03311
03312 protected:
03313     void initialize(timestamp_t initial_time) override;
03314     timestamp_t get_next() override;
03315
03316     void on_restart(timestamp_t restart_time) override;
03317
03318 private:
03319     std::minstd_rand0 generator;
03320     std::exponential_distribution<double> distr;
03321     timestamp_t accumulated;
03322     timestamp_t base_time;
03323 };
03324
03325 class TT_API GammaSignalGenerator : public SignalGeneratorBase {
03326 public:
03337     GammaSignalGenerator(TimeTaggerBase *tagger, double alpha, double beta, channel_t base_channel =

```



```

CHANNEL_UNUSED,
03338         int32_t seed = -1);
03339 ~GammaSignalGenerator();
03340
03341 protected:
03342 void initialize(timestamp_t initial_time) override;
03343 timestamp_t get_next() override;
03344
03345 void on_restart(timestamp_t restart_time) override;
03346
03347 private:
03348 std::minstd_rand0 generator;
03349 std::gamma_distribution<double> distr;
03350 timestamp_t accumulated;
03351 timestamp_t base_time;
03352 };
03353
03354 class TT_API PatternSignalGenerator : public SignalGeneratorBase {
03355 public:
03356 PatternSignalGenerator(TimeTaggerBase *tagger, std::vector<timestamp_t> sequence, bool repeat =
03357 false,
03358 timestamp_t start_delay = 0, timestamp_t spacing = 0, channel_t base_channel
03359 = CHANNEL_UNUSED);
03360 ~PatternSignalGenerator();
03361
03362 protected:
03363 void initialize(timestamp_t initial_time) override;
03364 timestamp_t get_next() override;
03365
03366 void on_restart(timestamp_t restart_time) override;
03367
03368 private:
03369 std::vector<timestamp_t> sequence;
03370 bool repeat;
03371 int64_t index;
03372 timestamp_t base_time;
03373 timestamp_t accumulated;
03374 timestamp_t start_delay;
03375 timestamp_t spacing;
03376 };
03377
03378 class SimSignalSplitterImpl;
03379 class TT_API SimSignalSplitter : public IteratorBase {
03380 public:
03381 SimSignalSplitter(TimeTaggerBase *tagger, channel_t input_channel, double ratio = 0.5, int32_t seed
03382 = -1);
03383 ~SimSignalSplitter();
03384
03385 std::vector<channel_t> getChannels();
03386 channel_t getLeftChannel();
03387 channel_t getRightChannel();
03388
03389 protected:
03390 bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
03391 override;
03392
03393 private:
03394 friend class SimSignalSplitterImpl;
03395 std::unique_ptr<SimSignalSplitterImpl> impl;
03396 };
03397
03398 class TT_API TransformEfficiency : public IteratorBase {
03399 public:
03400 TransformEfficiency(TimeTaggerBase *tagger, channel_t input_channel, double efficiency, bool copy =
03401 false,
03402 int32_t seed = -1);
03403 ~TransformEfficiency();
03404
03405 channel_t getChannel();
03406
03407 protected:
03408 bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
03409 override;
03410
03411 private:
03412 std::vector<Tag> mirror;
03413
03414 const channel_t input_channel;
03415 const channel_t output_channel;
03416
03417 const uint32_t limit;
03418 std::unique_ptr<PRBS> generator;
03419 };
03420
03421 class TT_API TransformGaussianBroadening : public IteratorBase {

```

```

03458 public:
03477     TransformGaussianBroadening(TimeTaggerBase *tagger, channel_t input_channel, double
standard_deviation,
03478                                     bool copy = false, int32_t seed = -1);
03479
03480     ~TransformGaussianBroadening();
03481
03482     channel_t getChannel();
03483
03484 protected:
03485     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
03486
03487 private:
03488     std::vector<Tag> mirror;
03489
03490     const channel_t input_channel;
03491     const channel_t output_channel;
03492
03493     std::minstd_rand0 generator;
03494     std::normal_distribution<double> distr;
03495
03496     bool overflow_state_on{};
03497
03498     std::vector<Tag> accumulated_tags;
03499     timestamp_t delay{};
03500     std::deque<Tag> delayed_tags;
03501 };
03502
03503 class TT_API TransformDeadtime : public IteratorBase {
03504 public:
03520     TransformDeadtime(TimeTaggerBase *tagger, channel_t input_channel, double deadtime, bool copy =
false);
03521
03522     ~TransformDeadtime();
03523
03524     channel_t getChannel();
03525
03526 protected:
03527     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
03528
03529 private:
03530     std::vector<Tag> mirror;
03531
03532     const channel_t input_channel;
03533     const channel_t output_channel;
03534
03535     timestamp_t deadtime{};
03536     timestamp_t last_gen_event{};
03537 };
03538
03539 class TT_API TransformCrosstalk : public IteratorBase {
03540 public:
03557     TransformCrosstalk(TimeTaggerBase *tagger, channel_t input_channel, channel_t relay_input_channel,
double delay,
03558                                     double tau, bool copy = false);
03559
03560     ~TransformCrosstalk();
03561
03562     channel_t getChannel();
03563
03564 protected:
03565     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
override;
03566
03567 private:
03568     std::vector<Tag> mirror;
03569
03570     const channel_t input_channel;
03571     const channel_t relay_input_channel;
03572     const channel_t output_channel;
03573
03574     double delay{};
03575     double tau{};
03576     double accumulated_delay{};
03577     timestamp_t last_timestamp{};
03578     std::deque<Tag> delayed_tags;
03579     bool overflow_state_on{};
03580 };
03581
03582 class TT_API SimDetector {
03583 public:
03596     SimDetector(TimeTaggerBase *tagger, channel_t input_channel, double efficiency = 1.0, double
darkcount_rate = 0.0,
03597                                     double jitter = 0, double deadtime = 0.0, int32_t seed = -1);
03598

```

```

03599 ~SimDetector();
03600
03601 channel_t getChannel();
03602
03603 private:
03604     channel_t output_channel;
03605     std::unique_ptr<TransformEfficiency> efficiency_meas;
03606     std::unique_ptr<ExponentialSignalGenerator> added_noise_meas;
03607     std::unique_ptr<TransformGaussianBroadening> jitter_meas;
03608     std::unique_ptr<TransformDeadtime> deadtime_meas;
03609 };
03610
03611 class TT_API SimLifetime : public IteratorBase {
03612 public:
03623     SimLifetime(TimeTaggerBase *tagger, channel_t input_channel, double lifetime, double emission_rate =
03624         0.1,
03625         int32_t seed = -1);
03626     ~SimLifetime();
03627     channel_t getChannel();
03628
03629     void registerLifetimeReactor(channel_t trigger_channel, std::vector<double> lifetimes, bool repeat);
03630
03631     void registerEmissionReactor(channel_t trigger_channel, std::vector<double> emissions, bool repeat);
03632
03633 protected:
03634     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
03635         override;
03636
03637 private:
03638     std::vector<Tag> mirror;
03639
03640     const channel_t input_channel;
03641     const channel_t output_channel;
03642
03643     std::minstd_rand0 generator;
03644     std::exponential_distribution<double> lifetime_distr;
03645     std::poisson_distribution<uint32_t> emission_distr;
03646     std::vector<Tag> accumulated_tags;
03647     bool overflow_state_on{};
03648
03649     // Reactors.
03650     bool has_reactor{};
03651
03652     std::vector<double> reactor_lifetimes;
03653     channel_t reactor_trigger_lifetimes;
03654     bool repeat_lifetimes;
03655     size_t current_index_lifetimes;
03656
03657     std::vector<double> reactor_emissions;
03658     channel_t reactor_trigger_emissions;
03659     bool repeat_emissions;
03660     size_t current_index_emissions;
03661 };
03662
03663 class PhotonNumberImpl;
03664 class TT_API PhotonNumber : public IteratorBase {
03665 public:
03666     PhotonNumber(TimeTaggerBase *tagger, channel_t trigger_ch, channel_t signal_start_ch, channel_t
03667         signal_stop_ch,
03668         double slope, std::vector<double> x_intercepts, timestamp_t dead_time);
03669
03670     ~PhotonNumber();
03671
03672     std::vector<channel_t> const &getChannels() const;
03673
03674 protected:
03675     bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t end_time)
03676         override;
03677     void clear_impl() override;
03678
03679 private:
03680     friend class PhotonNumberImpl;
03681     std::unique_ptr<PhotonNumberImpl> impl;
03682 };
03683
03684 } // namespace Experimental
03685
03686 #endif /* TT_ITERATORS_H_ */

```

## 10.3 TimeTagger.h File Reference

```
#include <atomic>
#include <condition_variable>
#include <cstdint>
#include <functional>
#include <limits>
#include <list>
#include <map>
#include <memory>
#include <mutex>
#include <set>
#include <stdexcept>
#include <stdint.h>
#include <string>
#include <unordered_set>
#include <vector>
```

Include dependency graph for TimeTagger.h:



### Classes

- struct [SoftwareClockState](#)
- class [CustomLogger](#)  
*Helper class for setLogger.*
- class [TimeTaggerBase](#)  
*Basis interface for all Time Tagger classes.*
- class [TimeTaggerVirtual](#)  
*virtual Time Tagger based on dump files*
- class [TimeTaggerNetwork](#)  
*network Time Tagger client.*
- class [TimeTagger](#)  
*backend for the TimeTagger.*
- struct [Tag](#)  
*a single event on a channel*
- class [OrderedBarrier](#)  
*Helper for implementing parallel measurements.*
- class [OrderedBarrier::OrderInstance](#)  
*Internal object for serialization.*
- class [OrderedPipeline](#)  
*Helper for implementing parallel measurements.*
- class [IteratorBase](#)  
*Base class for all iterators.*
- class [IteratorBase::AbortError](#)  
*A custom runtime error thrown by the abort call. This can be caught and handled by measurement classes, including CustomMeasurement, to perform actions within the abortion process.*

## Macros

- #define `TT_API` `__declspec(dllimport)`
- #define `timestamp_t` `long long`  
*The type for all timestamps used in the Time Tagger suite, always in picoseconds.*
- #define `channel_t` `int`  
*The type for storing a channel identifier.*
- #define `TIMETAGGER_VERSION` `"2.17.4"`  
*The version of this software suite.*
- #define `GET_DATA_1D`(function\_name, type, argout, attribute) attribute void function\_name(std::function<type \*(size\_t)> argout)  
*This are the default wrapper functions without any overloads.*
- #define `GET_DATA_1D_OP1`(function\_name, type, argout, optional\_type, optional\_name, optional\_default, attribute) attribute void function\_name(std::function<type \*(size\_t)> argout, optional\_type optional\_name = optional\_default)
- #define `GET_DATA_1D_OP2`(function\_name, type, argout, optional\_type, optional\_name, optional\_default, optional\_type2, optional\_name2, optional\_default2, attribute)
- #define `GET_DATA_2D`(function\_name, type, argout, attribute) attribute void function\_name(std::function<type \*(size\_t, size\_t)> argout)
- #define `GET_DATA_2D_OP1`(function\_name, type, argout, optional\_type, optional\_name, optional\_default, attribute)
- #define `GET_DATA_2D_OP2`(function\_name, type, argout, optional\_type, optional\_name, optional\_default, optional\_type2, optional\_name2, optional\_default2, attribute)
- #define `GET_DATA_3D`(function\_name, type, argout, attribute) attribute void function\_name(std::function<type \*(size\_t, size\_t, size\_t)> argout)
- #define `LogMessage`(level, ...) `LogBase`(level, \_\_FILE\_\_, \_\_LINE\_\_, false, \_\_VA\_ARGS\_\_);
- #define `ErrorLog`(...) `LogMessage`(`LOGGER_ERROR`, \_\_VA\_ARGS\_\_);
- #define `WarningLog`(...) `LogMessage`(`LOGGER_WARNING`, \_\_VA\_ARGS\_\_);
- #define `InfoLog`(...) `LogMessage`(`LOGGER_INFO`, \_\_VA\_ARGS\_\_);
- #define `LogMessageSuppressed`(level, ...) `LogBase`(level, \_\_FILE\_\_, \_\_LINE\_\_, true, \_\_VA\_ARGS\_\_);
- #define `ErrorLogSuppressed`(...) `LogMessageSuppressed`(`LOGGER_ERROR`, \_\_VA\_ARGS\_\_);
- #define `WarningLogSuppressed`(...) `LogMessageSuppressed`(`LOGGER_WARNING`, \_\_VA\_ARGS\_\_);
- #define `InfoLogSuppressed`(...) `LogMessageSuppressed`(`LOGGER_INFO`, \_\_VA\_ARGS\_\_);

## Typedefs

- typedef void(\* `logger_callback`) (`LogLevel` level, std::string msg)
- using `_Iterator` = `IteratorBase`

## Enumerations

- enum class `Resolution` { `Standard` = 0 , `HighResA` = 1 , `HighResB` = 2 , `HighResC` = 3 }  
*This enum selects the high resolution mode of the Time Tagger series.*
- enum class `ChannelEdge` : `int32_t` {  
`NoFalling` = 1 << 0 , `NoRising` = 1 << 1 , `NoStandard` = 1 << 2 , `NoHighRes` = 1 << 3 ,  
`All` = 0 , `Rising` = 1 , `Falling` = 2 , `HighResAll` = 4 ,  
`HighResRising` = 4 | 1 , `HighResFalling` = 4 | 2 , `StandardAll` = 8 , `StandardRising` = 8 | 1 ,  
`StandardFalling` = 8 | 2 }  
*Enum for filtering the channel list returned by `getChannelList`.*
- enum class `FpgaLinkInterface` { `SFPP_10GE` , `QSFP_40GE` }  
*Enum for selecting the fpga link output interface.*
- enum `LogLevel` { `LOGGER_ERROR` = 40 , `LOGGER_WARNING` = 30 , `LOGGER_INFO` = 10 }
- enum class `AccessMode` { `Listen` = 0 , `Control` = 2 , `SynchronousControl` = 3 }

- enum class `LanguageUsed` : `std::uint32_t` {  
`Cpp` = 0 , `Python` , `Csharp` , `Matlab` ,  
`Labview` , `Mathematica` , `Unknown` = 255 }
- enum class `FrontendType` : `std::uint32_t` {  
`Undefined` = 0 , `WebApp` , `Firefly` , `Pyro5RPC` ,  
`UserFrontend` }
- enum class `UsageStatisticsStatus` { `Disabled` , `Collecting` , `CollectingAndUploading` }

## Functions

- `TT_API std::string getVersion ()`  
*Get the version of the `TimeTagger` cxx backend.*
- `TT_API TimeTagger * createTimeTagger (std::string serial="", Resolution resolution=Resolution::Standard)`  
*default constructor factory.*
- `TT_API TimeTaggerVirtual * createTimeTaggerVirtual ()`  
*default constructor factory for the `createTimeTaggerVirtual` class.*
- `TT_API TimeTaggerNetwork * createTimeTaggerNetwork (std::string address="localhost:41101")`  
*default constructor factory for the `TimeTaggerNetwork` class.*
- `TT_API void setCustomBitFileName (const std::string &bitFileName)`  
*set path and filename of the bitfile to be loaded into the FPGA*
- `TT_API bool freeTimeTagger (TimeTaggerBase *tagger)`  
*free a copy of a `TimeTagger` reference.*
- `TT_API std::vector< std::string > scanTimeTagger ()`  
*fetches a list of all available `TimeTagger` serials.*
- `TT_API std::string getTimeTaggerServerInfo (std::string address="localhost:41101")`  
*connect to a `TimeTagger` server.*
- `TT_API std::vector< std::string > scanTimeTaggerServers ()`  
*scan the local network for running time tagger servers.*
- `TT_API std::string getTimeTaggerModel (const std::string &serial)`
- `TT_API void setTimeTaggerChannelNumberScheme (int scheme)`  
*Configure the numbering scheme for new `TimeTagger` objects.*
- `TT_API int getTimeTaggerChannelNumberScheme ()`  
*Fetch the currently configured global numbering scheme.*
- `TT_API bool hasTimeTaggerVirtualLicense ()`  
*Check if a license for the `TimeTaggerVirtual` is available.*
- `TT_API void flashLicense (const std::string &serial, const std::string &license)`  
*Update the license on the device.*
- `TT_API std::string extractDeviceLicense (const std::string &license)`  
*Converts binary license to JSON.*
- `TT_API logger_callback setLogger (logger_callback callback)`  
*Sets the notifier callback which is called for each log message.*
- `TT_API void LogBase (LogLevel level, const char *file, int line, bool suppressed, const char *fmt,...)`  
*Raise a new log message. Please use the `XXXLog` macro instead.*
- `TT_API void checkSystemLibraries ()`  
*Checks the MSVCP and okFrontPanel system library if they match the expected versions.*
- `TT_API bool operator== (Tag const &a, Tag const &b)`
- `TT_API void setLanguageInfo (std::uint32_t pw, LanguageUsed language, std::string version)`  
*sets the language being used currently for usage statistics system.*
- `TT_API void setFrontend (FrontendType frontend)`  
*sets the frontend being used currently for usage statistics system.*
- `TT_API void setUsageStatisticsStatus (UsageStatisticsStatus new_status)`

- sets the status of the usage statistics system.*
- [TT\\_API UsageStatisticsStatus](#) [getUsageStatisticsStatus](#) ()  
*gets the status of the usage statistics system.*
- [TT\\_API](#) [std::string](#) [getUsageStatisticsReport](#) ()  
*gets the current recorded data by the usage statistics system.*
- [TT\\_API](#) [void](#) [mergeStreamFiles](#) (const [std::string](#) &output\_filename, const [std::vector](#)< [std::string](#) > &input\_filenames, const [std::vector](#)< [int](#) > &channel\_offsets, const [std::vector](#)< [timestamp\\_t](#) > &time\_offsets, bool overlap\_only)  
*merges several tag streams.*

## Variables

- [constexpr](#) [channel\\_t](#) [CHANNEL\\_UNUSED](#) = -134217728  
*Constant for unused channel.*
- [constexpr](#) [channel\\_t](#) [CHANNEL\\_UNUSED\\_OLD](#) = -1
- [constexpr](#) [int](#) [TT\\_CHANNEL\\_NUMBER\\_SCHEME\\_AUTO](#) = 0  
*Allowed values for [setTimeTaggerChannelNumberScheme\(\)](#).*
- [constexpr](#) [int](#) [TT\\_CHANNEL\\_NUMBER\\_SCHEME\\_ZERO](#) = 1
- [constexpr](#) [int](#) [TT\\_CHANNEL\\_NUMBER\\_SCHEME\\_ONE](#) = 2
- [constexpr](#) [int](#) [TT\\_CHANNEL\\_NUMBER\\_SCHEME\\_DEFAULT](#) = 3
- [constexpr](#) [ChannelEdge](#) [TT\\_CHANNEL\\_RISING\\_AND\\_FALLING\\_EDGES](#) = [ChannelEdge::All](#)
- [constexpr](#) [ChannelEdge](#) [TT\\_CHANNEL\\_RISING\\_EDGES](#) = [ChannelEdge::Rising](#)
- [constexpr](#) [ChannelEdge](#) [TT\\_CHANNEL\\_FALLING\\_EDGES](#) = [ChannelEdge::Falling](#)

## 10.3.1 Macro Definition Documentation

### 10.3.1.1 channel\_t

```
#define channel_t int
```

The type for storing a channel identifier.

### 10.3.1.2 ErrorLog

```
#define ErrorLog(
    ... ) LogMessage(LOGGER\_ERROR, __VA_ARGS__);
```

### 10.3.1.3 ErrorLogSuppressed

```
#define ErrorLogSuppressed(
    ... ) LogMessageSuppressed(LOGGER\_ERROR, __VA_ARGS__);
```

### 10.3.1.4 GET\_DATA\_1D

```
#define GET_DATA_1D(
    function_name,
    type,
    argout,
    attribute ) attribute void function_name(std::function<type *(size_t)> argout)
```

This are the default wrapper functions without any overloadings.

### 10.3.1.5 GET\_DATA\_1D\_OP1

```
#define GET_DATA_1D_OP1(
    function_name,
    type,
    argout,
    optional_type,
    optional_name,
    optional_default,
    attribute ) attribute void function_name(std::function<type *(size_t)> argout,
optional_type optional_name = optional_default)
```

### 10.3.1.6 GET\_DATA\_1D\_OP2

```
#define GET_DATA_1D_OP2(
    function_name,
    type,
    argout,
    optional_type,
    optional_name,
    optional_default,
    optional_type2,
    optional_name2,
    optional_default2,
    attribute )
```

**Value:**

```
attribute void function_name(std::function<type *(size_t)> argout, optional_type optional_name =
optional_default, \
optional_type2 optional_name2 = optional_default2)
```

### 10.3.1.7 GET\_DATA\_2D

```
#define GET_DATA_2D(
    function_name,
    type,
    argout,
    attribute ) attribute void function_name(std::function<type *(size_t, size_t)>
argout)
```

### 10.3.1.8 GET\_DATA\_2D\_OP1

```
#define GET_DATA_2D_OP1(
    function_name,
    type,
    argout,
    optional_type,
    optional_name,
    optional_default,
    attribute )
```

**Value:**

```
attribute void function_name(std::function<type *(size_t, size_t)> argout,
\
optional_type optional_name = optional_default)
```



### 10.3.1.9 GET\_DATA\_2D\_OP2

```
#define GET_DATA_2D_OP2(
    function_name,
    type,
    argout,
    optional_type,
    optional_name,
    optional_default,
    optional_type2,
    optional_name2,
    optional_default2,
    attribute )
```

#### Value:

```
attribute void function_name(std::function<type *(size_t, size_t)> argout,
    \
    optional_type optional_name = optional_default,
    \
    optional_type2 optional_name2 = optional_default2)
```

### 10.3.1.10 GET\_DATA\_3D

```
#define GET_DATA_3D(
    function_name,
    type,
    argout,
    attribute ) attribute void function_name(std::function<type *(size_t, size_t,
size_t)> argout)
```

### 10.3.1.11 InfoLog

```
#define InfoLog(
    ... ) LogMessage(LOGGER\_INFO, __VA_ARGS__);
```

### 10.3.1.12 InfoLogSuppressed

```
#define InfoLogSuppressed(
    ... ) LogMessageSuppressed(LOGGER\_INFO, __VA_ARGS__);
```

### 10.3.1.13 LogMessage

```
#define LogMessage(
    level,
    ... ) LogBase(level, __FILE__, __LINE__, false, __VA_ARGS__);
```

### 10.3.1.14 LogMessageSuppressed

```
#define LogMessageSuppressed(
    level,
    ... ) LogBase(level, __FILE__, __LINE__, true, __VA_ARGS__);
```

#### 10.3.1.15 timestamp\_t

```
#define timestamp_t long long
```

The type for all timestamps used in the Time Tagger suite, always in picoseconds.

#### 10.3.1.16 TIMETAGGER\_VERSION

```
#define TIMETAGGER_VERSION "2.17.4"
```

The version of this software suite.

#### 10.3.1.17 TT\_API

```
#define TT_API __declspec(dllimport)
```

#### 10.3.1.18 WarningLog

```
#define WarningLog(  
    ... ) LogMessage(LOGGER\_WARNING, __VA_ARGS__);
```

#### 10.3.1.19 WarningLogSuppressed

```
#define WarningLogSuppressed(  
    ... ) LogMessageSuppressed(LOGGER\_WARNING, __VA_ARGS__);
```

### 10.3.2 Typedef Documentation

#### 10.3.2.1 \_Iterator

```
using \_Iterator = IteratorBase
```

#### 10.3.2.2 logger\_callback

```
typedef void(* logger_callback) (LogLevel level, std::string msg)
```

### 10.3.3 Enumeration Type Documentation

#### 10.3.3.1 AccessMode

```
enum class AccessMode [strong]
```

## Enumerator

Listen	
Control	
SynchronousControl	

### 10.3.3.2 ChannelEdge

```
enum class ChannelEdge : int32_t [strong]
```

Enum for filtering the channel list returned by getChannelList.

## Enumerator

NoFalling	
NoRising	
NoStandard	
NoHighRes	
All	
Rising	
Falling	
HighResAll	
HighResRising	
HighResFalling	
StandardAll	
StandardRising	
StandardFalling	

### 10.3.3.3 FpgaLinkInterface

```
enum class FpgaLinkInterface [strong]
```

Enum for selecting the fpga link output interface.

## Enumerator

SFPP_10GE	
QSFPP_40GE	

### 10.3.3.4 FrontendType

```
enum class FrontendType : std::uint32_t [strong]
```

## Enumerator

Undefined	
-----------	--

## Enumerator

WebApp	
Firefly	
Pyro5RPC	
UserFrontend	

**10.3.3.5 LanguageUsed**

```
enum class LanguageUsed : std::uint32_t [strong]
```

## Enumerator

Cpp	
Python	
Csharp	
Matlab	
Labview	
Mathematica	
Unknown	

**10.3.3.6 LogLevel**

```
enum LogLevel
```

## Enumerator

LOGGER_ERROR	
LOGGER_WARNING	
LOGGER_INFO	

**10.3.3.7 Resolution**

```
enum class Resolution [strong]
```

This enum selects the high resolution mode of the Time Tagger series.

If any high resolution mode is selected, the hardware will combine 2, 4 or even 8 input channels and average their timestamps. This results in a discretization jitter improvement of factor  $\sqrt{N}$  for  $N$  combined channels. The averaging is implemented before any filter, buffer or USB transmission. So all of those features are available with the averaged timestamps. Because of hardware limitations, only fixed combinations of channels are supported:

- HighResA: 1 : [1,2], 3 : [3,4], 5 : [5,6], 7 : [7,8], 10 : [10,11], 12 : [12,13], 14 : [14,15], 16 : [16,17], 9, 18
- HighResB: 1 : [1,2,3,4], 5 : [5,6,7,8], 10 : [10,11,12,13], 14 : [14,15,16,17], 9, 18
- HighResC: 5 : [1,2,3,4,5,6,7,8], 14 : [10,11,12,13,14,15,16,17], 9, 18 The inputs 9 and 18 are always available without averaging. The number of channels available will be limited to the number of channels licensed.

## Enumerator

Standard	
HighResA	
HighResB	
HighResC	

## 10.3.3.8 UsageStatisticsStatus

```
enum class UsageStatisticsStatus [strong]
```

## Enumerator

Disabled	
Collecting	
CollectingAndUploading	

## 10.3.4 Function Documentation

## 10.3.4.1 checkSystemLibraries()

```
TT_API void checkSystemLibraries ( )
```

Checks the MSVCP and okFrontPanel system library if they match the expected versions.

## 10.3.4.2 createTimeTagger()

```
TT_API TimeTagger * createTimeTagger (
    std::string serial = "",
    Resolution resolution = Resolution::Standard )
```

default constructor factory.

## Parameters

<i>serial</i>	serial number of FPGA board to use. if empty, the first board found is used.
<i>resolution</i>	enum for how many channels shall be grouped.

## See also

[Resolution](#) for details

## 10.3.4.3 createTimeTaggerNetwork()

```
TT_API TimeTaggerNetwork * createTimeTaggerNetwork (
    std::string address = "localhost:41101" )
```

default constructor factory for the [TimeTaggerNetwork](#) class.

#### Parameters

<i>address</i>	IP address of the server. Use hostname:port.
----------------	--

#### 10.3.4.4 createTimeTaggerVirtual()

```
TT_API TimeTaggerVirtual * createTimeTaggerVirtual ( )
```

default constructor factory for the createTimeTaggerVirtual class.

#### 10.3.4.5 extractDeviceLicense()

```
TT_API std::string extractDeviceLicense (
    const std::string & license )
```

Converts binary license to JSON.

#### Parameters

<i>license</i>	the binary license, encoded as a hexadecimal string
----------------	---

#### Returns

a JSON string containing the current device license

#### 10.3.4.6 flashLicense()

```
TT_API void flashLicense (
    const std::string & serial,
    const std::string & license )
```

Update the license on the device.

Updated license may be fetched by getRemoteLicense. The Time Tagger must not be instantiated while updating the license.

#### Parameters

<i>serial</i>	the serial of the device to update the license. Must not be empty
<i>license</i>	the binary license, encoded as a hexadecimal string

#### 10.3.4.7 freeTimeTagger()

```
TT_API bool freeTimeTagger (
    TimeTaggerBase * tagger )
```

free a copy of a [TimeTagger](#) reference.

#### Parameters

<i>tagger</i>	the <a href="#">TimeTagger</a> reference to free
---------------	--

#### 10.3.4.8 getTimeTaggerChannelNumberScheme()

```
TT_API int getTimeTaggerChannelNumberScheme ( )
```

Fetch the currently configured global numbering scheme.

Please see [setTimeTaggerChannelNumberScheme\(\)](#) for details. Please use [TimeTagger::getChannelNumberScheme\(\)](#) to query the actual used numbering scheme, this function here will just return the scheme a newly created [TimeTagger](#) object will use.

#### 10.3.4.9 getTimeTaggerModel()

```
TT_API std::string getTimeTaggerModel (
    const std::string & serial )
```

#### 10.3.4.10 getTimeTaggerServerInfo()

```
TT_API std::string getTimeTaggerServerInfo (
    std::string address = "localhost:41101" )
```

connect to a Time Tagger server.

#### Parameters

<i>address</i>	ip address or domain and port of the server hosting time tagger. Use hostname:port.
----------------	---

#### 10.3.4.11 getUsageStatisticsReport()

```
TT_API std::string getUsageStatisticsReport ( )
```

gets the current recorded data by the usage statistics system.

Use this function to see what data has been collected so far and what will be sent to Swabian Instruments if 'CollectingAndUploading' is enabled. All data is pseudonymous.

#### Note

if no data has been collected or due to a system error, the database was corrupted, it will return an error. else it will be a database in json format.

#### Returns

the current recorded data by the usage statistics system.

#### 10.3.4.12 `getUsageStatisticsStatus()`

```
TT_API UsageStatisticsStatus getUsageStatisticsStatus ( )
```

gets the status of the usage statistics system.

##### Returns

the current status of the usage statistics system.

#### 10.3.4.13 `getVersion()`

```
TT_API std::string getVersion ( )
```

Get the version of the [TimeTagger](#) cxx backend.

#### 10.3.4.14 `hasTimeTaggerVirtualLicense()`

```
TT_API bool hasTimeTaggerVirtualLicense ( )
```

Check if a license for the [TimeTaggerVirtual](#) is available.

#### 10.3.4.15 `LogBase()`

```
TT_API void LogBase (
    LogLevel level,
    const char * file,
    int line,
    bool suppressed,
    const char * fmt,
    ... )
```

Raise a new log message. Please use the XXXLog macro instead.

#### 10.3.4.16 `mergeStreamFiles()`

```
TT_API void mergeStreamFiles (
    const std::string & output_filename,
    const std::vector< std::string > & input_filenames,
    const std::vector< int > & channel_offsets,
    const std::vector< timestamp_t > & time_offsets,
    bool overlap_only )
```

merges several tag streams.

The function reads tags from several input streams, adjusts channel numbers and tag time as specified by 'channel\_offsets' and 'time\_offsets' respectively, and merges them to a single output stream. Throws if merge cannot be done.



## Parameters

<i>output_filename</i>	output stream file name, splitting is done as in 'FileWriter', with 1GB file size limit.
<i>input_filenames</i>	file names of input streams.
<i>channel_offsets</i>	offsets to shift channel numbers for corresponding input streams.
<i>time_offsets</i>	offsets to shift tag time for corresponding input streams.
<i>overlap_only</i>	specifies if only events in the time overlapping region of all input streams should be merged.

**10.3.4.17 operator==( )**

```
TT_API bool operator== (
    Tag const & a,
    Tag const & b )
```

**10.3.4.18 scanTimeTagger( )**

```
TT_API std::vector< std::string > scanTimeTagger ( )
```

fetches a list of all available [TimeTagger](#) serials.

This function may return serials blocked by other processes or already disconnected some milliseconds later.

**10.3.4.19 scanTimeTaggerServers( )**

```
TT_API std::vector< std::string > scanTimeTaggerServers ( )
```

scan the local network for running time tagger servers.

## Returns

a vector of strings of "ip\_address:port" for each active server in local network.

**10.3.4.20 setCustomBitFileName( )**

```
TT_API void setCustomBitFileName (
    const std::string & bitFileName )
```

set path and filename of the bitfile to be loaded into the FPGA

For debugging/development purposes the firmware loaded into the FPGA can be set manually with this function. To load the default bitfile set bitFileName = ""

## Parameters

<i>bitFileName</i>	custom bitfile to use for the FPGA.
--------------------	-------------------------------------

**10.3.4.21 setFrontend()**

```
TT_API void setFrontend (
    FrontendType frontend )
```

sets the frontend being used currently for usage statistics system.

**Parameters**

<i>frontend</i>	the frontend currently being used.
-----------------	------------------------------------

**10.3.4.22 setLanguageInfo()**

```
TT_API void setLanguageInfo (
    std::uint32_t pw,
    LanguageUsed language,
    std::string version )
```

sets the language being used currently for usage statistics system.

**Parameters**

<i>pw</i>	password for authorization to change the language.
<i>language</i>	programming language being used.
<i>version</i>	version of the programming language being used.

**10.3.4.23 setLogger()**

```
TT_API logger_callback setLogger (
    logger_callback callback )
```

Sets the notifier callback which is called for each log message.

If this function is called with nullptr, the default callback will be used.

**Returns**

The old callback

**10.3.4.24 setTimeTaggerChannelNumberScheme()**

```
TT_API void setTimeTaggerChannelNumberScheme (
    int scheme )
```

Configure the numbering scheme for new [TimeTagger](#) objects.

This function sets the numbering scheme for newly created [TimeTagger](#) objects. The default value is `_AUTO`.

Note: `TimeTagger` objects are cached internally, so the scheme should be set before the first call of `createTimeTagger()`.

`_ZERO` will typically allocate the channel numbers 0 to 7 for the 8 input channels. 8 to 15 will be allocated for the corresponding falling events.

`_ONE` will typically allocate the channel numbers 1 to 8 for the 8 input channels. -1 to -8 will be allocated for the corresponding falling events.

`_AUTO` will choose the scheme based on the hardware revision and so based on the printed label.

#### Parameters

<i>scheme</i>	new numbering scheme, must be <code>TT_CHANNEL_NUMBER_SCHEME_AUTO</code> , <code>TT_CHANNEL_NUMBER_SCHEME_ZERO</code> or <code>TT_CHANNEL_NUMBER_SCHEME_ONE</code>
---------------	--

#### 10.3.4.25 `setUsageStatisticsStatus()`

```
TT_API void setUsageStatisticsStatus (
    UsageStatisticsStatus new_status )
```

sets the status of the usage statistics system.

This functionality allows configuring the usage statistics system.

#### Parameters

<i>new_status</i>	new status of the usage statistics system.
-------------------	--

### 10.3.5 Variable Documentation

#### 10.3.5.1 `CHANNEL_UNUSED`

```
constexpr channel_t CHANNEL_UNUSED = -134217728 [constexpr]
```

Constant for unused channel.

Magic `channel_t` value to indicate an unused channel. So the iterators either have to disable this channel, or to choose a default one.

This value changed in version 2.1. The old value -1 aliases with falling events. The old value will still be accepted for now if the old numbering scheme is active.

#### 10.3.5.2 `CHANNEL_UNUSED_OLD`

```
constexpr channel_t CHANNEL_UNUSED_OLD = -1 [constexpr]
```

### 10.3.5.3 TT\_CHANNEL\_FALLING\_EDGES

```
constexpr ChannelEdge TT_CHANNEL_FALLING_EDGES = ChannelEdge::Falling [constexpr]
```

### 10.3.5.4 TT\_CHANNEL\_NUMBER\_SCHEME\_AUTO

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_AUTO = 0 [constexpr]
```

Allowed values for [setTimeTaggerChannelNumberScheme\(\)](#).

`_ZERO` will typically allocate the channel numbers 0 to 7 for the 8 input channels. 8 to 15 will be allocated for the corresponding falling events.

`_ONE` will typically allocate the channel numbers 1 to 8 for the 8 input channels. -1 to -8 will be allocated for the corresponding falling events.

`_AUTO` will choose the scheme based on the hardware revision and so based on the printed label.

`_DEFAULT` will always pick `_ONE`, but it will yield a warning if `_AUTO` would have picked `_ZERO`.

### 10.3.5.5 TT\_CHANNEL\_NUMBER\_SCHEME\_DEFAULT

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_DEFAULT = 3 [constexpr]
```

### 10.3.5.6 TT\_CHANNEL\_NUMBER\_SCHEME\_ONE

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_ONE = 2 [constexpr]
```

### 10.3.5.7 TT\_CHANNEL\_NUMBER\_SCHEME\_ZERO

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_ZERO = 1 [constexpr]
```

### 10.3.5.8 TT\_CHANNEL\_RISING\_AND\_FALLING\_EDGES

```
constexpr ChannelEdge TT_CHANNEL_RISING_AND_FALLING_EDGES = ChannelEdge::All [constexpr]
```

### 10.3.5.9 TT\_CHANNEL\_RISING\_EDGES

```
constexpr ChannelEdge TT_CHANNEL_RISING_EDGES = ChannelEdge::Rising [constexpr]
```

## 10.4 TimeTagger.h

[Go to the documentation of this file.](#)

```

00001 /*
00002 This file is part of Time Tagger software defined digital data acquisition.
00003
00004 Copyright (C) 2011-2019 Swabian Instruments
00005 All Rights Reserved
00006
00007 Unauthorized copying of this file is strictly prohibited.
00008 */
00009
00010 #ifndef TIMETAGGER_H_
00011 #define TIMETAGGER_H_
00012
00013 #ifdef _MSC_VER
00014 #pragma warning(disable : 4251)
00015 #endif
00016
00017 #ifdef LIBTIMETAGGER_EXPORTS
00018 #ifdef _WIN32
00019 #define TT_API __declspec(dllexport)
00020 #else
00021 #define TT_API __attribute__((visibility("default")))
00022 #endif
00023 #else
00024 #if defined(__linux) || defined(SWIG) || defined(NOEXPORT)
00025 #define TT_API
00026 #else
00027 #define TT_API __declspec(dllimport)
00028 #endif
00029 #ifdef _DEBUG
00030 #pragma comment(lib, "TimeTaggerD")
00031 #else
00032 #pragma comment(lib, "TimeTagger")
00033 #endif
00034 #endif
00035
00036 #include <atomic>
00037 #include <condition_variable>
00038 #include <cstdint>
00039 #include <functional>
00040 #include <limits>
00041 #include <list>
00042 #include <map>
00043 #include <memory>
00044 #include <mutex>
00045 #include <set>
00046 #include <stdexcept>
00047 #include <stdint.h>
00048 #include <string>
00049 #include <unordered_set>
00050 #include <vector>
00051
00070 class IteratorBase;
00071 class IteratorBaseListNode;
00072 class TimeTagger;
00073 class TimeTaggerBase;
00074 class TimeTaggerNetwork;
00075 class TimeTaggerRunner;
00076 class TimeTaggerVirtual;
00077
00079 #define timestamp_t long long
00080
00082 #define channel_t int
00083
00084 #ifndef SWIG
00086 #define TIMETAGGER_VERSION "2.17.4"
00087 #endif
00088
00092 TT_API std::string getVersion();
00093
00103 constexpr channel_t CHANNEL_UNUSED = -134217728;
00104 constexpr channel_t CHANNEL_UNUSED_OLD = -1;
00105
00119 constexpr int TT_CHANNEL_NUMBER_SCHEME_AUTO = 0;
00120 constexpr int TT_CHANNEL_NUMBER_SCHEME_ZERO = 1;
00121 constexpr int TT_CHANNEL_NUMBER_SCHEME_ONE = 2;
00122 constexpr int TT_CHANNEL_NUMBER_SCHEME_DEFAULT = 3;
00123
00124 #ifndef TIMETAGGER_NO_WRAPPER
00128 #define GET_DATA_1D(function_name, type, argout, attribute)
00129 \
    attribute void function_name(std::function<type *(size_t)> argout)
00130 #define GET_DATA_1D_Op1(function_name, type, argout, optional_type, optional_name, optional_default,
    attribute)
    \

```

```

00131 attribute void function_name(std::function<type *(size_t)> argout, optional_type optional_name =
00132 optional_default)
00132 #define GET_DATA_1D_OP2(function_name, type, argout, optional_type, optional_name, optional_default,
00133 optional_type2, \
00134 \
00134 \ attribute void function_name(std::function<type *(size_t)> argout, optional_type optional_name =
00135 optional_default, \
00135 \ optional_type2 optional_name2 = optional_default2)
00136 #define GET_DATA_2D(function_name, type, argout, attribute)
00137 \ attribute void function_name(std::function<type *(size_t, size_t)> argout)
00138 #define GET_DATA_2D_OP1(function_name, type, argout, optional_type, optional_name, optional_default,
00139 attribute) \
00139 \ attribute void function_name(std::function<type *(size_t, size_t)> argout,
00140 \
00140 \ optional_type optional_name = optional_default)
00141 #define GET_DATA_2D_OP2(function_name, type, argout, optional_type, optional_name, optional_default,
00142 optional_type2, \
00142 \ optional_name2, optional_default2, attribute)
00143 \ attribute void function_name(std::function<type *(size_t, size_t)> argout,
00144 \
00144 \ optional_type optional_name = optional_default,
00145 \
00145 \ optional_type2 optional_name2 = optional_default2)
00146 #define GET_DATA_3D(function_name, type, argout, attribute)
00147 \ attribute void function_name(std::function<type *(size_t, size_t, size_t)> argout)
00148 #endif
00149
00163 enum class Resolution { Standard = 0, HighResA = 1, HighResB = 2, HighResC = 3 };
00164
00168 enum class ChannelEdge : int32_t {
00169 // Bitwise filters, shall not be exported to wrapped languages
00170 #ifndef SWIG
00171 NoFalling = 1 << 0,
00172 NoRising = 1 << 1,
00173 NoStandard = 1 << 2,
00174 NoHighRes = 1 << 3,
00175 #endif
00176
00177 All = 0,
00178 Rising = 1,
00179 Falling = 2,
00180 HighResAll = 4,
00181 HighResRising = 4 | 1,
00182 HighResFalling = 4 | 2,
00183 StandardAll = 8,
00184 StandardRising = 8 | 1,
00185 StandardFalling = 8 | 2
00186 };
00187 constexpr ChannelEdge TT_CHANNEL_RISING_AND_FALLING_EDGES = ChannelEdge::All;
00188 constexpr ChannelEdge TT_CHANNEL_RISING_EDGES = ChannelEdge::Rising;
00189 constexpr ChannelEdge TT_CHANNEL_FALLING_EDGES = ChannelEdge::Falling;
00190
00191 struct SoftwareClockState {
00192 // configuration state
00193 timestamp_t clock_period;
00194 channel_t input_channel;
00195 channel_t ideal_clock_channel;
00196 double averaging_periods;
00197 bool enabled;
00198
00199 // runtime information
00200 bool is_locked;
00201 uint32_t error_counter;
00202 timestamp_t last_ideal_clock_event;
00203 double period_error; // in picoseconds
00204 double phase_error_estimation; // in picoseconds, including TDC discretization error
00205 };
00206
00210 enum class FpgaLinkInterface {
00211 SFPP_10GE,
00212 QSFP_40GE,
00213 };
00214
00221 TT_API TimeTagger *createTimeTagger(std::string serial = "", Resolution resolution =
00222 Resolution::Standard);
00226 TT_API TimeTaggerVirtual *createTimeTaggerVirtual();
00227
00233 TT_API TimeTaggerNetwork *createTimeTaggerNetwork(std::string address = "localhost:41101");
00234
00243 TT_API void setCustomBitFileName(const std::string &bitFileName);
00244
00250 TT_API bool freeTimeTagger(TimeTaggerBase *tagger);

```

```

00251
00257 TT_API std::vector<std::string> scanTimeTagger();
00258
00265 TT_API std::string getTimeTaggerServerInfo(std::string address = "localhost:41101");
00266
00272 TT_API std::vector<std::string> scanTimeTaggerServers();
00273
00274 /*
00275  * \brief returns the model name of the Time Tagger identified by the serial number.
00276  *
00277  * \param serial the Time Tagger serial number to query
00278  */
00279 TT_API std::string getTimeTaggerModel(const std::string &serial);
00280
00301 TT_API void setTimeTaggerChannelNumberScheme(int scheme);
00302
00310 TT_API int getTimeTaggerChannelNumberScheme();
00311
00315 TT_API bool hasTimeTaggerVirtualLicense();
00316
00326 TT_API void flashLicense(const std::string &serial, const std::string &license);
00327
00334 TT_API std::string extractDeviceLicense(const std::string &license);
00335
00336 // log values are taken from https://docs.python.org/3/library/logging.html
00337 enum LogLevel { LOGGER_ERROR = 40, LOGGER_WARNING = 30, LOGGER_INFO = 10 };
00338 typedef void (*logger_callback)(LogLevel level, std::string msg);
00339
00347 TT_API logger_callback setLogger(logger_callback callback);
00348
00352 TT_API void LogBase(LogLevel level, const char *file, int line, bool suppressed, const char *fmt, ...)
00353 #ifdef __GNUC__
00354     __attribute__((format(printf, 5, 6)))
00355 #endif
00356 ;
00357 #define LogMessage(level, ...) LogBase(level, __FILE__, __LINE__, false, __VA_ARGS__);
00358 #define ErrorLog(...) LogMessage(LOGGER_ERROR, __VA_ARGS__);
00359 #define WarningLog(...) LogMessage(LOGGER_WARNING, __VA_ARGS__);
00360 #define InfoLog(...) LogMessage(LOGGER_INFO, __VA_ARGS__);
00361
00362 // This suppressed methods are used when the log may contain private/confidential data and we
00363 // don't want the usage statistics system to record such data.
00364 #define LogMessageSuppressed(level, ...) LogBase(level, __FILE__, __LINE__, true, __VA_ARGS__);
00365 #define ErrorLogSuppressed(...) LogMessageSuppressed(LOGGER_ERROR, __VA_ARGS__);
00366 #define WarningLogSuppressed(...) LogMessageSuppressed(LOGGER_WARNING, __VA_ARGS__);
00367 #define InfoLogSuppressed(...) LogMessageSuppressed(LOGGER_INFO, __VA_ARGS__);
00368
00370 class TT_API CustomLogger {
00371 public:
00372     CustomLogger();
00373     virtual ~CustomLogger();
00374
00375     void enable();
00376     void disable();
00377     virtual void Log(int level, const std::string &msg) = 0;
00378
00379 private:
00380     static void LogCallback(LogLevel level, std::string msg);
00381     static CustomLogger *instance;
00382     static std::mutex instance_mutex;
00383 };
00384
00388 TT_API void checkSystemLibraries();
00389
00390 class ClientNetworkStream;
00391 class TT_API TimeTaggerBase {
00392     friend class IteratorBase;
00393     friend class TimeTaggerProxy;
00394     friend class TimeTaggerRunner;
00395     friend class ClientNetworkStream;
00396
00397 public:
00400     virtual unsigned int getFence(bool alloc_fence = true) = 0;
00401
00402     virtual bool waitForFence(unsigned int fence, int64_t timeout = -1) = 0;
00403
00404     virtual bool sync(int64_t timeout = -1) = 0;
00405
00406     virtual channel_t getInvertedChannel(channel_t channel) = 0;
00407
00408     virtual bool isUnusedChannel(channel_t channel) = 0;
00409
00410     typedef std::function<void(IteratorBase *)> IteratorCallback;
00411     typedef std::map<IteratorBase *, IteratorCallback> IteratorCallbackMap;
00412
00413     virtual void runSynchronized(const IteratorCallbackMap &callbacks, bool block = true) = 0;
00414

```

```

00484     virtual std::string getConfiguration() = 0;
00485
00500     virtual void setInputDelay(channel_t channel, timestamp_t delay) = 0;
00501
00513     virtual void setDelayHardware(channel_t channel, timestamp_t delay) = 0;
00514
00532     virtual void setDelaySoftware(channel_t channel, timestamp_t delay) = 0;
00533
00542     virtual timestamp_t getInputDelay(channel_t channel) = 0;
00543
00552     virtual timestamp_t getDelaySoftware(channel_t channel) = 0;
00553
00562     virtual timestamp_t getDelayHardware(channel_t channel) = 0;
00563
00577     virtual timestamp_t setDeadtime(channel_t channel, timestamp_t deadtime) = 0;
00578
00587     virtual timestamp_t getDeadtime(channel_t channel) = 0;
00588
00597     virtual void setTestSignal(channel_t channel, bool enabled) = 0;
00598
00607     virtual void setTestSignal(std::vector<channel_t> channel, bool enabled) = 0;
00608
00614     virtual bool getTestSignal(channel_t channel) = 0;
00615
00635     virtual void setSoftwareClock(channel_t input_channel, double input_frequency = 10e6, double
    averaging_periods = 1000,
00636                                     bool wait_until_locked = true) = 0;
00637
00643     virtual void disableSoftwareClock() = 0;
00644
00650     virtual SoftwareClockState getSoftwareClockState() = 0;
00651
00658     virtual long long getOverflows() = 0;
00659
00665     virtual void clearOverflows() = 0;
00666
00672     virtual long long getOverflowsAndClear() = 0;
00673
00674 protected:
00675     TimeTaggerBase() {}
00676
00683     virtual ~TimeTaggerBase(){};
00684
00685     // Non Copyable
00686     TimeTaggerBase(const TimeTaggerBase &) = delete;
00687     TimeTaggerBase &operator=(const TimeTaggerBase &) = delete;
00688
00689     // Used by IteratorBase to add itself
00690     virtual std::shared_ptr<IteratorBaseListNode> addIterator(IteratorBase *it) = 0;
00691
00692     // Used by IteratorBase to specify when it's being deleted.
00693     virtual void freeIterator(IteratorBase *it) = 0;
00694
00695     // allocate a new virtual output channel
00696     virtual channel_t getNewVirtualChannel() = 0;
00697
00698     // free a virtual channel being used.
00699     virtual void freeVirtualChannel(channel_t channel) = 0;
00700
00709     virtual void registerChannel(channel_t channel) = 0;
00710     virtual void registerChannel(std::set<channel_t> channels) = 0;
00711
00717     virtual void unregisterChannel(channel_t channel) = 0;
00718     virtual void unregisterChannel(std::set<channel_t> channels) = 0;
00719
00720     // Used by proxy time tagger to add itself as a dependent tagger.
00721     virtual void addChild(TimeTaggerBase *child) = 0;
00722
00723     // Used by proxy time tagger to remove itself as a dependent tagger.
00724     virtual void removeChild(TimeTaggerBase *child) = 0;
00725
00726     // Used by a proxy time tagger to allow its parent to release it and its dependencies.
00727     virtual void release() = 0;
00728 };
00729
00739 class TT_API TimeTaggerVirtual : virtual public TimeTaggerBase {
00740 public:
00753     virtual uint64_t replay(const std::string &file, timestamp_t begin = 0, timestamp_t duration = -1,
00754                             bool queue = true) = 0;
00755
00761     virtual void stop() = 0;
00762
00769     virtual void reset() = 0;
00770
00783     virtual bool waitForCompletion(uint64_t ID = 0, int64_t timeout = -1) = 0;
00784
00795     virtual void setReplaySpeed(double speed) = 0;

```



```

00796
00804     virtual double getReplaySpeed() = 0;
00805
00819     virtual void setConditionalFilter(std::vector<channel_t> trigger, std::vector<channel_t> filtered) =
0;
00820
00827     virtual void clearConditionalFilter() = 0;
00828
00834     virtual std::vector<channel_t> getConditionalFilterTrigger() = 0;
00835
00841     virtual std::vector<channel_t> getConditionalFilterFiltered() = 0;
00842
00848     virtual std::vector<channel_t> getChannelList() = 0;
00849 };
00850
00851 enum class AccessMode { Listen = 0, Control = 2, SynchronousControl = 3 };
00852
00862 class TT_API TimeTaggerNetwork : virtual public TimeTaggerBase {
00863 public:
00869     virtual bool isConnected() = 0;
00870
00877     virtual void setTriggerLevel(channel_t channel, double voltage) = 0;
00878
00884     virtual double getTriggerLevel(channel_t channel) = 0;
00885
00900     virtual void setConditionalFilter(std::vector<channel_t> trigger, std::vector<channel_t> filtered,
00901                                     bool hardwareDelayCompensation = true) = 0;
00902
00909     virtual void clearConditionalFilter() = 0;
00910
00916     virtual std::vector<channel_t> getConditionalFilterTrigger() = 0;
00917
00923     virtual std::vector<channel_t> getConditionalFilterFiltered() = 0;
00924
00934     virtual void setTestSignalDivider(int divider) = 0;
00935
00939     virtual int getTestSignalDivider() = 0;
00940
00946     virtual bool getTestSignal(channel_t channel) = 0;
00947
00958     virtual void setDelayClient(channel_t channel, timestamp_t time) = 0;
00959
00968     virtual timestamp_t getDelayClient(channel_t channel) = 0;
00969
00980     virtual timestamp_t getHardwareDelayCompensation(channel_t channel) = 0;
00981
00990     virtual void setNormalization(std::vector<channel_t> channels, bool state) = 0;
00991
01000     virtual bool getNormalization(channel_t channel) = 0;
01001
01010     virtual void setHardwareBufferSize(int size) = 0;
01011
01019     virtual int getHardwareBufferSize() = 0;
01020
01033     virtual void setStreamBlockSize(int max_events, int max_latency) = 0;
01034     virtual int getStreamBlockSizeEvents() = 0;
01035     virtual int getStreamBlockSizeLatency() = 0;
01036
01054     virtual void setEventDivider(channel_t channel, unsigned int divider) = 0;
01055
01064     virtual unsigned int getEventDivider(channel_t channel) = 0;
01065
01069     virtual std::string getSerial() = 0;
01070
01074     virtual std::string getModel() = 0;
01075
01081     virtual int getChannelNumberScheme() = 0;
01082
01086     virtual std::vector<double> getDACRange() = 0;
01087
01104     virtual std::vector<channel_t> getChannelList(ChannelEdge type = ChannelEdge::All) = 0;
01105
01109     virtual timestamp_t getPsPerClock() = 0;
01110
01115     virtual std::string getPcbVersion() = 0;
01116
01127     virtual std::string getFirmwareVersion() = 0;
01128
01132     virtual std::string getSensorData() = 0;
01133
01146     virtual void setLED(uint32_t bitmask) = 0;
01147
01152     virtual std::string getDeviceLicense() = 0;
01153
01158     virtual void setSoundFrequency(uint32_t freq_hz) = 0;
01159
01165     virtual void setTimeTaggerNetworkStreamCompression(bool active) = 0;

```

```

01166
01167     virtual long long getOverflowsClient() = 0;
01168     virtual void clearOverflowsClient() = 0;
01169     virtual long long getOverflowsAndClearClient() = 0;
01170
01171     virtual void setInputImpedanceHigh(channel_t channel, bool high_impedance) = 0;
01172
01173     virtual bool getInputImpedanceHigh(channel_t channel) = 0;
01174
01175     virtual void setInputHysteresis(channel_t channel, int value) = 0;
01176
01177     virtual int getInputHysteresis(channel_t channel) = 0;
01178 };
01179
01180 class TT_API TimeTagger : virtual public TimeTaggerBase {
01181 public:
01182     virtual void reset() = 0;
01183
01184     virtual bool isChannelRegistered(channel_t chan) = 0;
01185
01186     virtual void setTestSignalDivider(int divider) = 0;
01187
01188     virtual int getTestSignalDivider() = 0;
01189
01190     virtual void xtra_setAuxOutSignal(int channel, int divider, double duty_cycle = 0.5) = 0;
01191
01192     virtual int xtra_getAuxOutSignalDivider(int channel) = 0;
01193
01194     virtual double xtra_getAuxOutSignalDutyCycle(int channel) = 0;
01195
01196     virtual void xtra_setAuxOut(int channel, bool enabled) = 0;
01197
01198     virtual bool xtra_getAuxOut(int channel) = 0;
01199
01200     virtual void xtra_setFanSpeed(double percentage = -1) = 0;
01201
01202     virtual void setTriggerLevel(channel_t channel, double voltage) = 0;
01203
01204     virtual double getTriggerLevel(channel_t channel) = 0;
01205
01206     virtual double xtra_measureTriggerLevel(channel_t channel) = 0;
01207
01208     virtual timestamp_t getHardwareDelayCompensation(channel_t channel) = 0;
01209
01210     virtual void setInputMux(channel_t channel, int mux_mode) = 0;
01211
01212     virtual int getInputMux(channel_t channel) = 0;
01213
01214     virtual void setConditionalFilter(std::vector<channel_t> trigger, std::vector<channel_t> filtered,
01215                                     bool hardwareDelayCompensation = true) = 0;
01216
01217     virtual void clearConditionalFilter() = 0;
01218
01219     virtual std::vector<channel_t> getConditionalFilterTrigger() = 0;
01220
01221     virtual std::vector<channel_t> getConditionalFilterFiltered() = 0;
01222
01223     virtual void setNormalization(std::vector<channel_t> channels, bool state) = 0;
01224
01225     virtual bool getNormalization(channel_t channel) = 0;
01226
01227     virtual void setHardwareBufferSize(int size) = 0;
01228
01229     virtual int getHardwareBufferSize() = 0;
01230
01231     virtual void setStreamBlockSize(int max_events, int max_latency) = 0;
01232     virtual int getStreamBlockSizeEvents() = 0;
01233     virtual int getStreamBlockSizeLatency() = 0;
01234
01235     virtual void setEventDivider(channel_t channel, unsigned int divider) = 0;
01236
01237     virtual unsigned int getEventDivider(channel_t channel) = 0;
01238
01239     GET_DATA_1D(autoCalibration, double, array_out, virtual) = 0;
01240
01241     virtual std::string getSerial() = 0;
01242
01243     virtual std::string getModel() = 0;
01244
01245     virtual int getChannelNumberScheme() = 0;
01246
01247     virtual std::vector<double> getDACRange() = 0;
01248
01249     GET_DATA_2D(getDistributionCount, uint64_t, array_out, virtual) = 0;
01250
01251     GET_DATA_2D(getDistributionPSecs, double, array_out, virtual) = 0;
01252
01253
01254

```

```

01541     virtual std::vector<channel_t> getChannelList(ChannelEdge type = ChannelEdge::All) = 0;
01542
01546     virtual timestamp_t getPsPerClock() = 0;
01547
01552     virtual std::string getPcbVersion() = 0;
01553
01564     virtual std::string getFirmwareVersion() = 0;
01565
01575     virtual void xtra_setClockSource(int source) = 0;
01576
01589     virtual int xtra_getClockSource() = 0;
01590
01598     virtual void xtra_setClockAutoSelect(bool enabled) = 0;
01599
01607     virtual bool xtra_getClockAutoSelect() = 0;
01608
01616     virtual void xtra_setClockOut(bool enabled) = 0;
01617
01621     virtual std::string getSensorData() = 0;
01622
01635     virtual void setLED(uint32_t bitmask) = 0;
01636
01644     virtual void disableLEDs(bool disabled) = 0;
01645
01650     virtual std::string getDeviceLicense() = 0;
01651
01657     virtual uint32_t factoryAccess(uint32_t pw, uint32_t addr, uint32_t data, uint32_t mask, bool use_wb
= false) = 0;
01658
01664     virtual void setSoundFrequency(uint32_t freq_hz) = 0;
01665
01675     virtual void enableFpgaLink(std::vector<channel_t> channels, std::string destination_mac,
FpgaLinkInterface link_interface = FpgaLinkInterface::SFPP_10GE,
01676                                     bool exclusive = false) = 0;
01677
01678     virtual void disableFpgaLink() = 0;
01682
01683     virtual void startServer(AccessMode access_mode, std::vector<channel_t> channels =
std::vector<channel_t>(),
01691                                     uint32_t port = 41101) = 0;
01692
01693     virtual bool isServerRunning() = 0;
01699
01700     virtual void stopServer() = 0;
01706
01712     virtual void setTimeTaggerNetworkStreamCompression(bool active) = 0;
01713
01722     virtual void setInputImpedanceHigh(channel_t channel, bool high_impedance) = 0;
01723
01730     virtual bool getInputImpedanceHigh(channel_t channel) = 0;
01731
01741     virtual void setInputHysteresis(channel_t channel, int value) = 0;
01742
01749     virtual int getInputHysteresis(channel_t channel) = 0;
01750
01764     virtual void xtra_setAvgRisingFalling(channel_t channel, bool enable) = 0;
01765
01772     virtual bool xtra_getAvgRisingFalling(channel_t channel) = 0;
01773
01787     virtual void xtra_setHighPrioChannel(channel_t channel, bool enable) = 0;
01788
01795     virtual bool xtra_getHighPrioChannel(channel_t channel) = 0;
01796
01804     virtual void updateBMCFirmware(const std::string &firmware) = 0;
01805 };
01806
01815 struct TT_API Tag {
01827     enum class Type : unsigned char {
01828         TimeTag = 0,
01829         Error = 1,
01830         OverflowBegin = 2,
01831         OverflowEnd = 3,
01832         MissedEvents = 4
01833     } type{Type::TimeTag};
01834
01840     char reserved{};
01841
01850     unsigned short missed_events{};
01851
01853     channel_t channel{};
01854
01856     timestamp_t time{};
01857
01858     Tag() {}
01859     Tag(timestamp_t ts, channel_t ch, Type type = Type::TimeTag) : type{type}, channel{ch}, time{ts} {}
01860     Tag(Type type, char reserved, unsigned short missed_events, channel_t ch, timestamp_t ts)
01861         : type{type}, reserved{reserved}, missed_events{missed_events}, channel{ch}, time{ts} {}

```

```

01862 };
01863
01864 TT_API bool operator==(Tag const &a, Tag const &b);
01865
01866 class TT_API OrderedBarrier {
01867 public:
01870 class TT_API OrderInstance {
01871 public:
01872     OrderInstance();
01873     OrderInstance(OrderedBarrier *parent, uint64_t instance_id);
01874     ~OrderInstance();
01875     void sync();
01876     void release();
01877
01878 private:
01879     friend class OrderedBarrier;
01880
01881     OrderedBarrier *parent{};
01882     bool obtained{};
01883     uint64_t instance_id{};
01884 };
01885
01886 OrderedBarrier();
01887 ~OrderedBarrier();
01888
01889 OrderInstance queue();
01890 void waitUntilFinished();
01891
01892 private:
01893     friend class OrderInstance;
01894
01895     void release(uint64_t index);
01896     void obtain(uint64_t index);
01897
01898     uint64_t accumulator{};
01899     uint64_t current_state{};
01900     std::mutex inner_mutex;
01901     std::condition_variable cv;
01902 };
01903
01904 class TT_API OrderedPipeline {
01905 public:
01906     OrderedPipeline();
01907     ~OrderedPipeline();
01908
01909 private:
01910     friend class IteratorBase;
01911
01912     bool initialized = false;
01913     std::list<OrderedBarrier>::iterator stage;
01914 };
01915
01916 class TT_API IteratorBase {
01917 friend class TimeTaggerRunner;
01918 friend class TimeTaggerProxy;
01919 friend class SynchronizedMeasurements;
01920 friend class IteratorTest;
01921
01922 private:
01923     // Abstract class
01924     IteratorBase() = delete;
01925
01926     // Non Copyable
01927     IteratorBase(const IteratorBase &) = delete;
01928     IteratorBase &operator=(const IteratorBase &) = delete;
01929     void clearWithoutLock();
01930
01931 protected:
01932     IteratorBase(TimeTaggerBase *tagger, std::string base_type_ = "IteratorBase", std::string
01933     extra_info_ = "");
01934
01935 public:
01936     virtual ~IteratorBase();
01937
01938     void start();
01939
01940     void startFor(timestamp_t capture_duration, bool clear = true);
01941
01942     bool waitUntilFinished(int64_t timeout = -1);
01943
01944     void stop();
01945
01946     void clear();
01947
01948     void abort();
01949
01950     bool isRunning();

```

```

02010
02016     timestamp_t getCaptureDuration();
02017
02023     std::string getConfiguration();
02024
02030     class AbortError : public std::runtime_error {
02031     public:
02032         AbortError(const std::string &what_arg) : std::runtime_error(what_arg){};
02033         ~AbortError(){};
02034     };
02035
02036 protected:
02044     void registerChannel(channel_t channel);
02045
02051     void unregisterChannel(channel_t channel);
02052
02056     channel_t getNewVirtualChannel();
02057
02061     void finishInitialization();
02062
02070     virtual void clear_impl(){};
02071
02077     virtual void on_start(){};
02078
02084     virtual void on_stop(){};
02085
02095     void lock();
02096
02104     void unlock();
02105
02114     OrderedBarrier::OrderInstance parallelize(OrderedPipeline &pipeline);
02115
02125     std::unique_lock<std::mutex> getLock();
02126
02141     virtual bool next_impl(std::vector<Tag> &incoming_tags, timestamp_t begin_time, timestamp_t
end_time) = 0;
02142
02150     void finish_running();
02151
02153     std::set<channel_t> channels_registered;
02154
02156     bool running;
02157
02159     bool autostart;
02160
02162     TimeTaggerBase *tagger;
02163
02165     timestamp_t capture_duration;
02167     timestamp_t pre_capture_duration;
02168
02169     // to abort measurement;
02170     std::atomic<bool> aborting;
02171     // This call shall be placed in every next_impl loop to allow single threaded measurements to abort.
02172     // It will be inlined wherever header is included
02173     void checkForAbort() {
02174         if (aborting) {
02175             on_abort();
02176         }
02177     };
02178     // Overload for multithreaded measurements
02179     template <typename T> void checkForAbort(T callback) {
02180         if (aborting) {
02181             callback();
02182             on_abort();
02183         }
02184     };
02185
02186 private:
02187     struct TelemetryData {
02188         uint64_t duration;
02189         bool is_on;
02190     };
02191
02192     void next(std::unique_lock<std::mutex> &lock, std::vector<Tag> &incoming_tags, timestamp_t
begin_time,
02193             timestamp_t end_time, uint32_t fence, TelemetryData &telem_data);
02194
02195     void pre_stop();
02196     void on_abort();
02197     std::shared_ptr<IteratorBaseListNode> iter;
02198     timestamp_t max_capture_duration; // capture duration at which the .stop() method will be called, <0
for infinity
02199     std::mutex pre_stop_mutex;
02200     uint32_t min_fence;
02201     std::unordered_set<channel_t> virtual_channels;
02202     const std::string base_type;
02203     const std::string extra_info;

```

```

02204     uint64_t id{};
02205     bool initialized{};
02206     uint64_t clear_tick{};
02207 };
02208 using _Iterator = IteratorBase;
02209
02210 enum class LanguageUsed : std::uint32_t {
02211     Cpp = 0,
02212     Python,
02213     Csharp,
02214     Matlab,
02215     Labview,
02216     Mathematica,
02217     // Add more languages/Platforms
02218     Unknown = 255,
02219 };
02220
02221 enum class FrontendType : std::uint32_t {
02222     Undefined = 0,
02223     WebApp,
02224     Firefly,
02225     Pyro5RPC,
02226     UserFrontend,
02227 };
02228
02236 TT_API void setLanguageInfo(std::uint32_t pw, LanguageUsed language, std::string version);
02237
02243 TT_API void setFrontend(FrontendType frontend);
02244
02245 enum class UsageStatisticsStatus {
02246     Disabled, // User Opted out
02247     Collecting, // User enabled it to collect for debug purpose
02248     CollectingAndUploading, // User gave their consent to collect and upload
02249 };
02250
02258 TT_API void setUsageStatisticsStatus(UsageStatisticsStatus new_status);
02259
02265 TT_API UsageStatisticsStatus getUsageStatisticsStatus();
02266
02278 TT_API std::string getUsageStatisticsReport();
02279
02295 TT_API void mergeStreamFiles(const std::string &output_filename, const std::vector<std::string>
&input_filenames,
02296                               const std::vector<int> &channel_offsets, const std::vector<timestamp_t>
&time_offsets,
02297                               bool overlap_only);
02298
02299 #endif /* TIMETAGGER_H_ */

```