

TimeTagger

2.14.0.0

Generated by Doxygen 1.8.17



<b>1 TimeTagger</b>	<b>1</b>
<b>2 Module Index</b>	<b>3</b>
2.1 Modules . . . . .	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 Module 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 . . . . .	16
7.2.1 Detailed Description . . . . .	16
7.3 Event counting . . . . .	17
7.3.1 Detailed Description . . . . .	17
7.4 Time histograms . . . . .	18
7.4.1 Detailed Description . . . . .	18
7.5 Fluorescence-lifetime imaging (FLIM) . . . . .	19
7.5.1 Detailed Description . . . . .	19
7.6 Time-tag-streaming . . . . .	20
7.6.1 Detailed Description . . . . .	20
7.7 Helper classes . . . . .	21
7.7.1 Detailed Description . . . . .	21
7.8 Virtual Channels . . . . .	22
7.8.1 Detailed Description . . . . .	22
<b>8 Namespace Documentation</b>	<b>23</b>
8.1 Experimental Namespace Reference . . . . .	23
<b>9 Class Documentation</b>	<b>25</b>
9.1 Coincidence Class Reference . . . . .	25
9.1.1 Detailed Description . . . . .	26
9.1.2 Constructor & Destructor Documentation . . . . .	26
9.1.2.1 Coincidence() . . . . .	26
9.1.3 Member Function Documentation . . . . .	26
9.1.3.1 getChannel() . . . . .	27

9.2 Coincidences Class Reference	27
9.2.1 Detailed Description	28
9.2.2 Constructor & Destructor Documentation	28
9.2.2.1 Coincidences()	28
9.2.2.2 ~Coincidences()	28
9.2.3 Member Function Documentation	28
9.2.3.1 getChannels()	28
9.2.3.2 next_impl()	29
9.2.3.3 setCoincidenceWindow()	29
9.3 Combiner Class Reference	29
9.3.1 Detailed Description	30
9.3.2 Constructor & Destructor Documentation	30
9.3.2.1 Combiner()	30
9.3.2.2 ~Combiner()	31
9.3.3 Member Function Documentation	31
9.3.3.1 clear_impl()	31
9.3.3.2 getChannel()	31
9.3.3.3 getChannelCounts()	31
9.3.3.4 getData()	32
9.3.3.5 next_impl()	32
9.4 ConstantFractionDiscriminator Class Reference	32
9.4.1 Detailed Description	33
9.4.2 Constructor & Destructor Documentation	34
9.4.2.1 ConstantFractionDiscriminator()	34
9.4.2.2 ~ConstantFractionDiscriminator()	34
9.4.3 Member Function Documentation	34
9.4.3.1 getChannels()	34
9.4.3.2 next_impl()	34
9.4.3.3 on_start()	35
9.5 Correlation Class Reference	35
9.5.1 Detailed Description	36
9.5.2 Constructor & Destructor Documentation	37
9.5.2.1 Correlation()	37
9.5.2.2 ~Correlation()	37
9.5.3 Member Function Documentation	37
9.5.3.1 clear_impl()	37
9.5.3.2 getData()	38
9.5.3.3 getDataNormalized()	38
9.5.3.4 getIndex()	38
9.5.3.5 next_impl()	38
9.6 CountBetweenMarkers Class Reference	39
9.6.1 Detailed Description	40

9.6.2 Constructor & Destructor Documentation	41
9.6.2.1 CountBetweenMarkers()	41
9.6.2.2 ~CountBetweenMarkers()	41
9.6.3 Member Function Documentation	41
9.6.3.1 clear_impl()	41
9.6.3.2 getBinWidths()	42
9.6.3.3 getData()	42
9.6.3.4 getIndex()	42
9.6.3.5 next_impl()	42
9.6.3.6 ready()	43
9.7 Counter Class Reference	43
9.7.1 Detailed Description	44
9.7.2 Constructor & Destructor Documentation	44
9.7.2.1 Counter()	44
9.7.2.2 ~Counter()	45
9.7.3 Member Function Documentation	45
9.7.3.1 clear_impl()	45
9.7.3.2 getData()	45
9.7.3.3 getDataNormalized()	46
9.7.3.4 getDataObject()	46
9.7.3.5 getDataTotalCounts()	46
9.7.3.6 getIndex()	46
9.7.3.7 next_impl()	47
9.7.3.8 on_start()	47
9.8 CounterData Class Reference	47
9.8.1 Detailed Description	48
9.8.2 Constructor & Destructor Documentation	48
9.8.2.1 ~CounterData()	48
9.8.3 Member Function Documentation	48
9.8.3.1 getChannels()	49
9.8.3.2 getData()	49
9.8.3.3 getDataNormalized()	49
9.8.3.4 getDataTotalCounts()	49
9.8.3.5 getIndex()	49
9.8.3.6 getOverflowMask()	49
9.8.3.7 getTime()	50
9.8.4 Member Data Documentation	50
9.8.4.1 dropped_bins	50
9.8.4.2 overflow	50
9.8.4.3 size	50
9.9 Countrate Class Reference	50
9.9.1 Detailed Description	51

9.9.2 Constructor & Destructor Documentation	51
9.9.2.1 Countrate()	51
9.9.2.2 ~Countrate()	52
9.9.3 Member Function Documentation	52
9.9.3.1 clear_impl()	52
9.9.3.2 getCountsTotal()	52
9.9.3.3 getData()	52
9.9.3.4 next_impl()	52
9.9.3.5 on_start()	53
9.10 CustomLogger Class Reference	53
9.10.1 Detailed Description	53
9.10.2 Constructor & Destructor Documentation	54
9.10.2.1 CustomLogger()	54
9.10.2.2 ~CustomLogger()	54
9.10.3 Member Function Documentation	54
9.10.3.1 disable()	54
9.10.3.2 enable()	54
9.10.3.3 Log()	54
9.11 CustomMeasurementBase Class Reference	55
9.11.1 Detailed Description	56
9.11.2 Constructor & Destructor Documentation	56
9.11.2.1 CustomMeasurementBase()	56
9.11.2.2 ~CustomMeasurementBase()	56
9.11.3 Member Function Documentation	56
9.11.3.1 _lock()	56
9.11.3.2 _unlock()	56
9.11.3.3 clear_impl()	56
9.11.3.4 finalize_init()	57
9.11.3.5 is_running()	57
9.11.3.6 next_impl()	57
9.11.3.7 next_impl_cs()	57
9.11.3.8 on_start()	58
9.11.3.9 on_stop()	58
9.11.3.10 register_channel()	58
9.11.3.11 stop_all_custom_measurements()	58
9.11.3.12 unregister_channel()	58
9.12 DelayedChannel Class Reference	59
9.12.1 Detailed Description	60
9.12.2 Constructor & Destructor Documentation	60
9.12.2.1 DelayedChannel() [1/2]	60
9.12.2.2 DelayedChannel() [2/2]	60
9.12.2.3 ~DelayedChannel()	61

9.12.3 Member Function Documentation	61
9.12.3.1 getChannel()	61
9.12.3.2 getChannels()	61
9.12.3.3 next_impl()	61
9.12.3.4 on_start()	62
9.12.3.5 setDelay()	62
9.13 Dump Class Reference	63
9.13.1 Detailed Description	63
9.13.2 Constructor & Destructor Documentation	64
9.13.2.1 Dump()	64
9.13.2.2 ~Dump()	64
9.13.3 Member Function Documentation	64
9.13.3.1 clear_impl()	64
9.13.3.2 next_impl()	64
9.13.3.3 on_start()	65
9.13.3.4 on_stop()	65
9.14 Event Struct Reference	65
9.14.1 Detailed Description	66
9.14.2 Member Data Documentation	66
9.14.2.1 state	66
9.14.2.2 time	66
9.15 EventGenerator Class Reference	66
9.15.1 Detailed Description	67
9.15.2 Constructor & Destructor Documentation	67
9.15.2.1 EventGenerator()	68
9.15.2.2 ~EventGenerator()	68
9.15.3 Member Function Documentation	68
9.15.3.1 clear_impl()	68
9.15.3.2 getChannel()	68
9.15.3.3 next_impl()	69
9.15.3.4 on_start()	69
9.16 Experimental::ExponentialSignalGenerator Class Reference	70
9.16.1 Constructor & Destructor Documentation	70
9.16.1.1 ExponentialSignalGenerator()	70
9.16.1.2 ~ExponentialSignalGenerator()	71
9.16.2 Member Function Documentation	71
9.16.2.1 get_next()	71
9.16.2.2 initialize()	71
9.16.2.3 on_restart()	71
9.17 FastBinning Class Reference	72
9.17.1 Detailed Description	72
9.17.2 Member Enumeration Documentation	72

9.17.2.1 Mode	72
9.17.3 Constructor & Destructor Documentation	73
9.17.3.1 FastBinning() [1/2]	73
9.17.3.2 FastBinning() [2/2]	73
9.17.4 Member Function Documentation	73
9.17.4.1 divide()	73
9.17.4.2 getMode()	73
9.18 FileReader Class Reference	73
9.18.1 Detailed Description	74
9.18.2 Constructor & Destructor Documentation	74
9.18.2.1 FileReader() [1/2]	74
9.18.2.2 FileReader() [2/2]	75
9.18.2.3 ~FileReader()	75
9.18.3 Member Function Documentation	75
9.18.3.1 getChannelList()	75
9.18.3.2 getConfiguration()	75
9.18.3.3 getData()	75
9.18.3.4 getDataRaw()	76
9.18.3.5 getLastMarker()	76
9.18.3.6 hasData()	76
9.19 FileWriter Class Reference	77
9.19.1 Detailed Description	78
9.19.2 Constructor & Destructor Documentation	78
9.19.2.1 FileWriter()	78
9.19.2.2 ~FileWriter()	78
9.19.3 Member Function Documentation	78
9.19.3.1 clear_impl()	78
9.19.3.2 getMaxFileSize()	79
9.19.3.3 getTotalEvents()	79
9.19.3.4 getTotalSize()	79
9.19.3.5 next_impl()	79
9.19.3.6 on_start()	80
9.19.3.7 on_stop()	80
9.19.3.8 setMarker()	80
9.19.3.9 setMaxFileSize()	81
9.19.3.10 split()	81
9.20 Flim Class Reference	81
9.20.1 Detailed Description	83
9.20.2 Constructor & Destructor Documentation	84
9.20.2.1 Flim()	84
9.20.2.2 ~Flim()	84
9.20.3 Member Function Documentation	84



9.20.3.1 clear_impl()	85
9.20.3.2 frameReady()	85
9.20.3.3 get_ready_index()	85
9.20.3.4 getCurrentFrame()	85
9.20.3.5 getCurrentFrameEx()	85
9.20.3.6 getCurrentFrameIntensity()	86
9.20.3.7 getFramesAcquired()	86
9.20.3.8 getIndex()	86
9.20.3.9 getReadyFrame()	86
9.20.3.10 getReadyFrameEx()	87
9.20.3.11 getReadyFrameIntensity()	87
9.20.3.12 getSummedFrames()	87
9.20.3.13 getSummedFramesEx()	88
9.20.3.14 getSummedFramesIntensity()	88
9.20.3.15 initialize()	89
9.20.3.16 on_frame_end()	89
9.20.4 Member Data Documentation	89
9.20.4.1 accum_diffs	89
9.20.4.2 back_frames	89
9.20.4.3 captured_frames	89
9.20.4.4 frame_begins	89
9.20.4.5 frame_ends	90
9.20.4.6 last_frame	90
9.20.4.7 pixels_completed	90
9.20.4.8 summed_frames	90
9.20.4.9 swap_chain_lock	90
9.20.4.10 total_frames	90
9.21 FlimAbstract Class Reference	91
9.21.1 Detailed Description	92
9.21.2 Constructor & Destructor Documentation	92
9.21.2.1 FlimAbstract()	92
9.21.2.2 ~FlimAbstract()	93
9.21.3 Member Function Documentation	93
9.21.3.1 clear_impl()	93
9.21.3.2 isAcquiring()	94
9.21.3.3 next_impl()	94
9.21.3.4 on_frame_end()	94
9.21.3.5 on_start()	95
9.21.3.6 process_tags()	95
9.21.4 Member Data Documentation	95
9.21.4.1 acquiring	95
9.21.4.2 acquisition_lock	95

9.21.4.3 binner	95
9.21.4.4 binwidth	95
9.21.4.5 click_channel	96
9.21.4.6 current_frame_begin	96
9.21.4.7 current_frame_end	96
9.21.4.8 data_base	96
9.21.4.9 finish_after_outputframe	96
9.21.4.10 frame	96
9.21.4.11 frame_acquisition	96
9.21.4.12 frame_begin_channel	96
9.21.4.13 frames_completed	97
9.21.4.14 initialized	97
9.21.4.15 n_bins	97
9.21.4.16 n_frame_average	97
9.21.4.17 n_pixels	97
9.21.4.18 pixel_acquisition	97
9.21.4.19 pixel_begin_channel	97
9.21.4.20 pixel_begins	97
9.21.4.21 pixel_end_channel	98
9.21.4.22 pixel_ends	98
9.21.4.23 pixels_processed	98
9.21.4.24 previous_starts	98
9.21.4.25 start_channel	98
9.21.4.26 ticks	98
9.21.4.27 time_window	98
9.22 FlimBase Class Reference	99
9.22.1 Detailed Description	100
9.22.2 Constructor & Destructor Documentation	100
9.22.2.1 FlimBase()	100
9.22.2.2 ~FlimBase()	101
9.22.3 Member Function Documentation	101
9.22.3.1 frameReady()	101
9.22.3.2 initialize()	101
9.22.3.3 on_frame_end()	101
9.22.4 Member Data Documentation	101
9.22.4.1 total_frames	101
9.23 FlimFrameInfo Class Reference	102
9.23.1 Detailed Description	102
9.23.2 Constructor & Destructor Documentation	102
9.23.2.1 ~FlimFrameInfo()	102
9.23.3 Member Function Documentation	102
9.23.3.1 getFrameNumber()	103

9.23.3.2	<a href="#">getHistograms()</a>	103
9.23.3.3	<a href="#">getIntensities()</a>	103
9.23.3.4	<a href="#">getPixelBegins()</a>	103
9.23.3.5	<a href="#">getPixelEnds()</a>	103
9.23.3.6	<a href="#">getPixelPosition()</a>	103
9.23.3.7	<a href="#">getSummedCounts()</a>	104
9.23.3.8	<a href="#">isValid()</a>	104
9.23.4	<a href="#">Member Data Documentation</a>	104
9.23.4.1	<a href="#">bins</a>	104
9.23.4.2	<a href="#">frame_number</a>	104
9.23.4.3	<a href="#">pixel_position</a>	104
9.23.4.4	<a href="#">pixels</a>	104
9.23.4.5	<a href="#">valid</a>	105
9.24	<a href="#">FrequencyMultiplier Class Reference</a>	105
9.24.1	<a href="#">Detailed Description</a>	106
9.24.2	<a href="#">Constructor &amp; Destructor Documentation</a>	106
9.24.2.1	<a href="#">FrequencyMultiplier()</a>	106
9.24.2.2	<a href="#">~FrequencyMultiplier()</a>	107
9.24.3	<a href="#">Member Function Documentation</a>	107
9.24.3.1	<a href="#">getChannel()</a>	107
9.24.3.2	<a href="#">getMultiplier()</a>	107
9.24.3.3	<a href="#">next_impl()</a>	107
9.25	<a href="#">FrequencyStability Class Reference</a>	108
9.25.1	<a href="#">Detailed Description</a>	108
9.25.2	<a href="#">Constructor &amp; Destructor Documentation</a>	109
9.25.2.1	<a href="#">FrequencyStability()</a>	109
9.25.2.2	<a href="#">~FrequencyStability()</a>	110
9.25.3	<a href="#">Member Function Documentation</a>	110
9.25.3.1	<a href="#">clear_impl()</a>	110
9.25.3.2	<a href="#">getDataObject()</a>	110
9.25.3.3	<a href="#">next_impl()</a>	110
9.25.3.4	<a href="#">on_start()</a>	111
9.26	<a href="#">FrequencyStabilityData Class Reference</a>	111
9.26.1	<a href="#">Detailed Description</a>	112
9.26.2	<a href="#">Constructor &amp; Destructor Documentation</a>	112
9.26.2.1	<a href="#">~FrequencyStabilityData()</a>	112
9.26.3	<a href="#">Member Function Documentation</a>	112
9.26.3.1	<a href="#">getADEV()</a>	112
9.26.3.2	<a href="#">getADEVScaled()</a>	112
9.26.3.3	<a href="#">getHDEV()</a>	113
9.26.3.4	<a href="#">getHDEVScaled()</a>	113
9.26.3.5	<a href="#">getMDEV()</a>	113

9.26.3.6	getSTDD()	113
9.26.3.7	getTau()	113
9.26.3.8	getTDEV()	113
9.26.3.9	getTraceFrequency()	114
9.26.3.10	getTraceFrequencyAbsolute()	114
9.26.3.11	getTraceIndex()	114
9.26.3.12	getTracePhase()	114
9.27	Experimental::GammaSignalGenerator Class Reference	115
9.27.1	Constructor & Destructor Documentation	115
9.27.1.1	GammaSignalGenerator()	115
9.27.1.2	~GammaSignalGenerator()	116
9.27.2	Member Function Documentation	116
9.27.2.1	get_next()	116
9.27.2.2	initialize()	116
9.27.2.3	on_restart()	116
9.28	GatedChannel Class Reference	117
9.28.1	Detailed Description	117
9.28.2	Constructor & Destructor Documentation	118
9.28.2.1	GatedChannel()	118
9.28.2.2	~GatedChannel()	118
9.28.3	Member Function Documentation	119
9.28.3.1	getChannel()	119
9.28.3.2	next_impl()	119
9.29	Experimental::GaussianSignalGenerator Class Reference	119
9.29.1	Constructor & Destructor Documentation	120
9.29.1.1	GaussianSignalGenerator()	120
9.29.1.2	~GaussianSignalGenerator()	121
9.29.2	Member Function Documentation	121
9.29.2.1	get_next()	121
9.29.2.2	initialize()	121
9.29.2.3	on_restart()	121
9.30	Histogram Class Reference	122
9.30.1	Detailed Description	122
9.30.2	Constructor & Destructor Documentation	123
9.30.2.1	Histogram()	123
9.30.2.2	~Histogram()	124
9.30.3	Member Function Documentation	124
9.30.3.1	clear_impl()	124
9.30.3.2	getData()	124
9.30.3.3	getIndex()	124
9.30.3.4	next_impl()	124
9.30.3.5	on_start()	125

9.31 Histogram2D Class Reference . . . . .	125
9.31.1 Detailed Description . . . . .	126
9.31.2 Constructor & Destructor Documentation . . . . .	127
9.31.2.1 Histogram2D() . . . . .	127
9.31.2.2 ~Histogram2D() . . . . .	127
9.31.3 Member Function Documentation . . . . .	127
9.31.3.1 clear_impl() . . . . .	127
9.31.3.2 getData() . . . . .	128
9.31.3.3 getIndex() . . . . .	128
9.31.3.4 getIndex_1() . . . . .	128
9.31.3.5 getIndex_2() . . . . .	128
9.31.3.6 next_impl() . . . . .	128
9.32 HistogramLogBins Class Reference . . . . .	129
9.32.1 Detailed Description . . . . .	130
9.32.2 Constructor & Destructor Documentation . . . . .	130
9.32.2.1 HistogramLogBins() . . . . .	131
9.32.2.2 ~HistogramLogBins() . . . . .	131
9.32.3 Member Function Documentation . . . . .	131
9.32.3.1 clear_impl() . . . . .	131
9.32.3.2 getBinEdges() . . . . .	132
9.32.3.3 getData() . . . . .	132
9.32.3.4 getDataNormalizedCountsPerPs() . . . . .	132
9.32.3.5 getDataNormalizedG2() . . . . .	132
9.32.3.6 next_impl() . . . . .	132
9.33 HistogramND Class Reference . . . . .	133
9.33.1 Detailed Description . . . . .	134
9.33.2 Constructor & Destructor Documentation . . . . .	134
9.33.2.1 HistogramND() . . . . .	134
9.33.2.2 ~HistogramND() . . . . .	134
9.33.3 Member Function Documentation . . . . .	135
9.33.3.1 clear_impl() . . . . .	135
9.33.3.2 getData() . . . . .	135
9.33.3.3 getIndex() . . . . .	135
9.33.3.4 next_impl() . . . . .	135
9.34 Iterator Class Reference . . . . .	136
9.34.1 Detailed Description . . . . .	137
9.34.2 Constructor & Destructor Documentation . . . . .	137
9.34.2.1 Iterator() . . . . .	137
9.34.2.2 ~Iterator() . . . . .	137
9.34.3 Member Function Documentation . . . . .	137
9.34.3.1 clear_impl() . . . . .	137
9.34.3.2 next() . . . . .	138

9.34.3.3 next_impl()	138
9.34.3.4 size()	138
9.35 IteratorBase Class Reference	139
9.35.1 Detailed Description	141
9.35.2 Constructor & Destructor Documentation	141
9.35.2.1 IteratorBase()	141
9.35.2.2 ~IteratorBase()	141
9.35.3 Member Function Documentation	141
9.35.3.1 clear()	142
9.35.3.2 clear_impl()	142
9.35.3.3 finish_running()	142
9.35.3.4 finishInitialization()	142
9.35.3.5 getCaptureDuration()	142
9.35.3.6 getConfiguration()	143
9.35.3.7 getLock()	143
9.35.3.8 getNewVirtualChannel()	143
9.35.3.9 isRunning()	143
9.35.3.10 lock()	144
9.35.3.11 next_impl()	144
9.35.3.12 on_start()	144
9.35.3.13 on_stop()	145
9.35.3.14 parallelize()	145
9.35.3.15 registerChannel()	145
9.35.3.16 start()	145
9.35.3.17 startFor()	146
9.35.3.18 stop()	146
9.35.3.19 unlock()	146
9.35.3.20 unregisterChannel()	146
9.35.3.21 waitUntilFinished()	147
9.35.4 Member Data Documentation	147
9.35.4.1 autostart	147
9.35.4.2 capture_duration	147
9.35.4.3 channels_registered	147
9.35.4.4 pre_capture_duration	148
9.35.4.5 running	148
9.35.4.6 tagger	148
9.36 Experimental::MarkovProcessGenerator Class Reference	148
9.36.1 Constructor & Destructor Documentation	149
9.36.1.1 MarkovProcessGenerator()	149
9.36.1.2 ~MarkovProcessGenerator()	150
9.36.2 Member Function Documentation	150
9.36.2.1 getChannel()	150

9.36.2.2 getChannels()	150
9.36.2.3 next_impl()	150
9.36.2.4 on_stop()	151
9.37 OrderedBarrier Class Reference	151
9.37.1 Detailed Description	151
9.37.2 Constructor & Destructor Documentation	151
9.37.2.1 OrderedBarrier()	151
9.37.2.2 ~OrderedBarrier()	152
9.37.3 Member Function Documentation	152
9.37.3.1 queue()	152
9.37.3.2 waitUntilFinished()	152
9.38 OrderedPipeline Class Reference	152
9.38.1 Detailed Description	152
9.38.2 Constructor & Destructor Documentation	152
9.38.2.1 OrderedPipeline()	153
9.38.2.2 ~OrderedPipeline()	153
9.39 OrderedBarrier::OrderInstance Class Reference	153
9.39.1 Detailed Description	153
9.39.2 Constructor & Destructor Documentation	153
9.39.2.1 OrderInstance() [1/2]	153
9.39.2.2 OrderInstance() [2/2]	154
9.39.2.3 ~OrderInstance()	154
9.39.3 Member Function Documentation	154
9.39.3.1 release()	154
9.39.3.2 sync()	154
9.40 Experimental::PatternSignalGenerator Class Reference	154
9.40.1 Constructor & Destructor Documentation	155
9.40.1.1 PatternSignalGenerator()	155
9.40.1.2 ~PatternSignalGenerator()	155
9.40.2 Member Function Documentation	156
9.40.2.1 get_next()	156
9.40.2.2 initialize()	156
9.40.2.3 on_restart()	156
9.41 Experimental::PoissonSignalGenerator Class Reference	156
9.41.1 Constructor & Destructor Documentation	157
9.41.1.1 PoissonSignalGenerator()	157
9.41.1.2 ~PoissonSignalGenerator()	157
9.41.2 Member Function Documentation	157
9.41.2.1 get_next()	158
9.41.2.2 initialize()	158
9.41.2.3 on_restart()	158
9.42 Sampler Class Reference	158

9.42.1 Detailed Description	159
9.42.2 Constructor & Destructor Documentation	159
9.42.2.1 Sampler()	159
9.42.2.2 ~Sampler()	160
9.42.3 Member Function Documentation	160
9.42.3.1 clear_impl()	160
9.42.3.2 getData()	160
9.42.3.3 getDataAsMask()	160
9.42.3.4 next_impl()	161
9.42.3.5 on_start()	161
9.43 Scope Class Reference	162
9.43.1 Detailed Description	162
9.43.2 Constructor & Destructor Documentation	163
9.43.2.1 Scope()	163
9.43.2.2 ~Scope()	164
9.43.3 Member Function Documentation	164
9.43.3.1 clear_impl()	164
9.43.3.2 getData()	164
9.43.3.3 getWindowSize()	164
9.43.3.4 next_impl()	164
9.43.3.5 ready()	165
9.43.3.6 triggered()	165
9.44 Experimental::SignalGeneratorBase Class Reference	165
9.44.1 Constructor & Destructor Documentation	166
9.44.1.1 SignalGeneratorBase()	166
9.44.1.2 ~SignalGeneratorBase()	166
9.44.2 Member Function Documentation	166
9.44.2.1 get_next()	167
9.44.2.2 getChannel()	167
9.44.2.3 initialize()	167
9.44.2.4 isProcessingFinished()	167
9.44.2.5 next_impl()	167
9.44.2.6 on_restart()	168
9.44.2.7 on_stop()	168
9.44.2.8 set_processing_finished()	168
9.44.3 Member Data Documentation	168
9.44.3.1 impl	169
9.45 Experimental::SimDetector Class Reference	169
9.45.1 Constructor & Destructor Documentation	169
9.45.1.1 SimDetector()	169
9.45.1.2 ~SimDetector()	170
9.45.2 Member Function Documentation	170



9.45.2.1	getChannel()	170
9.46	Experimental::SimLifetime Class Reference	170
9.46.1	Constructor & Destructor Documentation	171
9.46.1.1	SimLifetime()	171
9.46.1.2	~SimLifetime()	171
9.46.2	Member Function Documentation	171
9.46.2.1	getChannel()	171
9.46.2.2	next_impl()	172
9.46.2.3	registerEmissionReactor()	172
9.46.2.4	registerLifetimeReactor()	172
9.47	Experimental::SimSignalSplitter Class Reference	173
9.47.1	Constructor & Destructor Documentation	173
9.47.1.1	SimSignalSplitter()	173
9.47.1.2	~SimSignalSplitter()	174
9.47.2	Member Function Documentation	174
9.47.2.1	getChannels()	174
9.47.2.2	getLeftChannel()	174
9.47.2.3	getRightChannel()	174
9.47.2.4	next_impl()	174
9.48	SoftwareClockState Struct Reference	175
9.48.1	Member Data Documentation	175
9.48.1.1	averaging_periods	175
9.48.1.2	clock_period	176
9.48.1.3	enabled	176
9.48.1.4	error_counter	176
9.48.1.5	ideal_clock_channel	176
9.48.1.6	input_channel	176
9.48.1.7	is_locked	176
9.48.1.8	last_ideal_clock_event	176
9.48.1.9	period_error	176
9.48.1.10	phase_error_estimation	177
9.49	StartStop Class Reference	177
9.49.1	Detailed Description	178
9.49.2	Constructor & Destructor Documentation	178
9.49.2.1	StartStop()	178
9.49.2.2	~StartStop()	179
9.49.3	Member Function Documentation	179
9.49.3.1	clear_impl()	179
9.49.3.2	getData()	179
9.49.3.3	next_impl()	179
9.49.3.4	on_start()	180
9.50	SynchronizedMeasurements Class Reference	180

9.50.1 Detailed Description	181
9.50.2 Constructor & Destructor Documentation	181
9.50.2.1 SynchronizedMeasurements()	181
9.50.2.2 ~SynchronizedMeasurements()	181
9.50.3 Member Function Documentation	181
9.50.3.1 clear()	181
9.50.3.2 getTagger()	182
9.50.3.3 isRunning()	182
9.50.3.4 registerMeasurement()	182
9.50.3.5 runCallback()	182
9.50.3.6 start()	182
9.50.3.7 startFor()	183
9.50.3.8 stop()	183
9.50.3.9 unregisterMeasurement()	183
9.50.3.10 waitUntilFinished()	183
9.51 SyntheticSingleTag Class Reference	183
9.51.1 Detailed Description	184
9.51.2 Constructor & Destructor Documentation	184
9.51.2.1 SyntheticSingleTag()	184
9.51.2.2 ~SyntheticSingleTag()	185
9.51.3 Member Function Documentation	185
9.51.3.1 getChannel()	185
9.51.3.2 next_impl()	185
9.51.3.3 trigger()	186
9.52 Tag Struct Reference	186
9.52.1 Detailed Description	186
9.52.2 Member Enumeration Documentation	186
9.52.2.1 Type	187
9.52.3 Member Data Documentation	187
9.52.3.1 channel	187
9.52.3.2 missed_events	187
9.52.3.3 reserved	188
9.52.3.4 time	188
9.52.3.5 type	188
9.53 TimeDifferences Class Reference	188
9.53.1 Detailed Description	189
9.53.2 Constructor & Destructor Documentation	190
9.53.2.1 TimeDifferences()	190
9.53.2.2 ~TimeDifferences()	191
9.53.3 Member Function Documentation	191
9.53.3.1 clear_impl()	191
9.53.3.2 getCounts()	191

9.53.3.3	getData()	191
9.53.3.4	getHistogramIndex()	191
9.53.3.5	getIndex()	192
9.53.3.6	next_impl()	192
9.53.3.7	on_start()	192
9.53.3.8	ready()	193
9.53.3.9	setMaxCounts()	193
9.54	TimeDifferencesImpl< T > Class Template Reference	193
9.55	TimeDifferencesND Class Reference	193
9.55.1	Detailed Description	194
9.55.2	Constructor & Destructor Documentation	195
9.55.2.1	TimeDifferencesND()	195
9.55.2.2	~TimeDifferencesND()	196
9.55.3	Member Function Documentation	196
9.55.3.1	clear_impl()	196
9.55.3.2	getData()	196
9.55.3.3	getIndex()	196
9.55.3.4	next_impl()	196
9.55.3.5	on_start()	197
9.56	TimeTagger Class Reference	197
9.56.1	Detailed Description	200
9.56.2	Member Function Documentation	200
9.56.2.1	autoCalibration()	200
9.56.2.2	clearConditionalFilter()	201
9.56.2.3	disableFpgaLink()	201
9.56.2.4	disableLEDs()	201
9.56.2.5	enableFpgaLink()	201
9.56.2.6	factoryAccess()	202
9.56.2.7	getChannelList()	202
9.56.2.8	getChannelNumberScheme()	202
9.56.2.9	getConditionalFilterFiltered()	202
9.56.2.10	getConditionalFilterTrigger()	202
9.56.2.11	getDACRange()	203
9.56.2.12	getDeviceLicense()	203
9.56.2.13	getDistributionCount()	203
9.56.2.14	getDistributionPSecs()	203
9.56.2.15	getEventDivider()	203
9.56.2.16	getFirmwareVersion()	204
9.56.2.17	getHardwareBufferSize()	204
9.56.2.18	getHardwareDelayCompensation()	204
9.56.2.19	getInputHysteresis()	205
9.56.2.20	getInputImpedanceHigh()	205

9.56.2.21	getInputMux()	205
9.56.2.22	getModel()	207
9.56.2.23	getNormalization()	207
9.56.2.24	getPcbVersion()	207
9.56.2.25	getPsPerClock()	208
9.56.2.26	getSensorData()	208
9.56.2.27	getSerial()	208
9.56.2.28	getStreamBlockSizeEvents()	208
9.56.2.29	getStreamBlockSizeLatency()	208
9.56.2.30	getTestSignalDivider()	208
9.56.2.31	getTriggerLevel()	208
9.56.2.32	isChannelRegistered()	209
9.56.2.33	isServerRunning()	209
9.56.2.34	reset()	209
9.56.2.35	setConditionalFilter()	209
9.56.2.36	setEventDivider()	210
9.56.2.37	setHardwareBufferSize()	210
9.56.2.38	setInputHysteresis()	210
9.56.2.39	setInputImpedanceHigh()	211
9.56.2.40	setInputMux()	211
9.56.2.41	setLED()	212
9.56.2.42	setNormalization()	212
9.56.2.43	setSoundFrequency()	212
9.56.2.44	setStreamBlockSize()	212
9.56.2.45	setTestSignalDivider()	213
9.56.2.46	setTimeTaggerNetworkStreamCompression()	213
9.56.2.47	setTriggerLevel()	213
9.56.2.48	startServer()	215
9.56.2.49	stopServer()	215
9.56.2.50	xtra_getAuxOut()	215
9.56.2.51	xtra_getAuxOutSignalDivider()	216
9.56.2.52	xtra_getAuxOutSignalDutyCycle()	216
9.56.2.53	xtra_getClockAutoSelect()	216
9.56.2.54	xtra_getClockSource()	217
9.56.2.55	xtra_measureTriggerLevel()	217
9.56.2.56	xtra_setAuxOut()	217
9.56.2.57	xtra_setAuxOutSignal()	218
9.56.2.58	xtra_setClockAutoSelect()	218
9.56.2.59	xtra_setClockOut()	218
9.56.2.60	xtra_setClockSource()	220
9.57	TimeTaggerBase Class Reference	220
9.57.1	Detailed Description	222

9.57.2 Member Typedef Documentation	222
9.57.2.1 IteratorCallback	222
9.57.2.2 IteratorCallbackMap	222
9.57.3 Constructor & Destructor Documentation	222
9.57.3.1 TimeTaggerBase() [1/2]	223
9.57.3.2 ~TimeTaggerBase()	223
9.57.3.3 TimeTaggerBase() [2/2]	223
9.57.4 Member Function Documentation	223
9.57.4.1 addChild()	223
9.57.4.2 addIterator()	223
9.57.4.3 clearOverflows()	223
9.57.4.4 disableSoftwareClock()	224
9.57.4.5 freeIterator()	224
9.57.4.6 freeVirtualChannel()	224
9.57.4.7 getConfiguration()	224
9.57.4.8 getDeadtime()	224
9.57.4.9 getDelayHardware()	225
9.57.4.10 getDelaySoftware()	225
9.57.4.11 getFence()	225
9.57.4.12 getInputDelay()	226
9.57.4.13 getInvertedChannel()	226
9.57.4.14 getNewVirtualChannel()	227
9.57.4.15 getOverflows()	227
9.57.4.16 getOverflowsAndClear()	227
9.57.4.17 getSoftwareClockState()	227
9.57.4.18 getTestSignal()	227
9.57.4.19 isUnusedChannel()	228
9.57.4.20 operator=()	228
9.57.4.21 registerChannel() [1/2]	228
9.57.4.22 registerChannel() [2/2]	228
9.57.4.23 release()	228
9.57.4.24 removeChild()	229
9.57.4.25 runSynchronized()	229
9.57.4.26 setDeadtime()	229
9.57.4.27 setDelayHardware()	230
9.57.4.28 setDelaySoftware()	230
9.57.4.29 setInputDelay()	230
9.57.4.30 setSoftwareClock()	231
9.57.4.31 setTestSignal() [1/2]	231
9.57.4.32 setTestSignal() [2/2]	232
9.57.4.33 sync()	232
9.57.4.34 unregisterChannel() [1/2]	233

9.57.4.35 unregisterChannel() [2/2]	233
9.57.4.36 waitForFence()	233
9.58 TimeTaggerNetwork Class Reference	233
9.58.1 Detailed Description	236
9.58.2 Member Function Documentation	236
9.58.2.1 clearConditionalFilter()	236
9.58.2.2 clearOverflowsClient()	236
9.58.2.3 getChannelList()	236
9.58.2.4 getChannelNumberScheme()	236
9.58.2.5 getConditionalFilterFiltered()	237
9.58.2.6 getConditionalFilterTrigger()	237
9.58.2.7 getDACRange()	237
9.58.2.8 getDelayClient()	237
9.58.2.9 getDeviceLicense()	237
9.58.2.10 getEventDivider()	238
9.58.2.11 getFirmwareVersion()	238
9.58.2.12 getHardwareBufferSize()	238
9.58.2.13 getHardwareDelayCompensation()	239
9.58.2.14 getInputHysteresis()	239
9.58.2.15 getInputImpedanceHigh()	239
9.58.2.16 getModel()	240
9.58.2.17 getNormalization()	240
9.58.2.18 getOverflowsAndClearClient()	240
9.58.2.19 getOverflowsClient()	240
9.58.2.20 getPcbVersion()	241
9.58.2.21 getPsPerClock()	241
9.58.2.22 getSensorData()	241
9.58.2.23 getSerial()	241
9.58.2.24 getStreamBlockSizeEvents()	241
9.58.2.25 getStreamBlockSizeLatency()	241
9.58.2.26 getTestSignal()	241
9.58.2.27 getTestSignalDivider()	242
9.58.2.28 getTriggerLevel()	242
9.58.2.29 isConnected()	242
9.58.2.30 setConditionalFilter()	242
9.58.2.31 setDelayClient()	244
9.58.2.32 setEventDivider()	244
9.58.2.33 setHardwareBufferSize()	245
9.58.2.34 setInputHysteresis()	245
9.58.2.35 setInputImpedanceHigh()	245
9.58.2.36 setLED()	246
9.58.2.37 setNormalization()	246

9.58.2.38 setSoundFrequency()	246
9.58.2.39 setStreamBlockSize()	246
9.58.2.40 setTestSignalDivider()	247
9.58.2.41 setTimeTaggerNetworkStreamCompression()	247
9.58.2.42 setTriggerLevel()	247
9.59 TimeTaggerVirtual Class Reference	248
9.59.1 Detailed Description	249
9.59.2 Member Function Documentation	249
9.59.2.1 clearConditionalFilter()	249
9.59.2.2 getConditionalFilterFiltered()	249
9.59.2.3 getConditionalFilterTrigger()	249
9.59.2.4 getReplaySpeed()	249
9.59.2.5 replay()	250
9.59.2.6 reset()	250
9.59.2.7 setConditionalFilter()	250
9.59.2.8 setReplaySpeed()	251
9.59.2.9 stop()	251
9.59.2.10 waitForCompletion()	251
9.60 TimeTagStream Class Reference	252
9.60.1 Detailed Description	252
9.60.2 Constructor & Destructor Documentation	252
9.60.2.1 TimeTagStream()	253
9.60.2.2 ~TimeTagStream()	253
9.60.3 Member Function Documentation	253
9.60.3.1 clear_impl()	253
9.60.3.2 getCounts()	253
9.60.3.3 getData()	254
9.60.3.4 next_impl()	254
9.61 TimeTagStreamBuffer Class Reference	254
9.61.1 Detailed Description	255
9.61.2 Constructor & Destructor Documentation	255
9.61.2.1 ~TimeTagStreamBuffer()	255
9.61.3 Member Function Documentation	255
9.61.3.1 getChannels()	255
9.61.3.2 getEventTypes()	255
9.61.3.3 getMissedEvents()	255
9.61.3.4 getOverflows()	256
9.61.3.5 getTimestamps()	256
9.61.4 Member Data Documentation	256
9.61.4.1 hasOverflows	256
9.61.4.2 size	256
9.61.4.3 tGetData	256

9.61.4.4 tStart . . . . .	256
9.62 Experimental::TransformCrosstalk Class Reference . . . . .	257
9.62.1 Constructor & Destructor Documentation . . . . .	257
9.62.1.1 TransformCrosstalk() . . . . .	257
9.62.1.2 ~TransformCrosstalk() . . . . .	258
9.62.2 Member Function Documentation . . . . .	258
9.62.2.1 getChannel() . . . . .	258
9.62.2.2 next_impl() . . . . .	258
9.63 Experimental::TransformDeadtime Class Reference . . . . .	259
9.63.1 Constructor & Destructor Documentation . . . . .	259
9.63.1.1 TransformDeadtime() . . . . .	260
9.63.1.2 ~TransformDeadtime() . . . . .	260
9.63.2 Member Function Documentation . . . . .	260
9.63.2.1 getChannel() . . . . .	260
9.63.2.2 next_impl() . . . . .	260
9.64 Experimental::TransformEfficiency Class Reference . . . . .	261
9.64.1 Constructor & Destructor Documentation . . . . .	262
9.64.1.1 TransformEfficiency() . . . . .	262
9.64.1.2 ~TransformEfficiency() . . . . .	262
9.64.2 Member Function Documentation . . . . .	262
9.64.2.1 getChannel() . . . . .	262
9.64.2.2 next_impl() . . . . .	263
9.65 Experimental::TransformGaussianBroadening Class Reference . . . . .	263
9.65.1 Constructor & Destructor Documentation . . . . .	264
9.65.1.1 TransformGaussianBroadening() . . . . .	264
9.65.1.2 ~TransformGaussianBroadening() . . . . .	265
9.65.2 Member Function Documentation . . . . .	265
9.65.2.1 getChannel() . . . . .	265
9.65.2.2 next_impl() . . . . .	265
9.66 TriggerOnCountrate Class Reference . . . . .	265
9.66.1 Detailed Description . . . . .	267
9.66.2 Constructor & Destructor Documentation . . . . .	267
9.66.2.1 TriggerOnCountrate() . . . . .	267
9.66.2.2 ~TriggerOnCountrate() . . . . .	268
9.66.3 Member Function Documentation . . . . .	268
9.66.3.1 getChannelAbove() . . . . .	268
9.66.3.2 getChannelBelow() . . . . .	268
9.66.3.3 getChannels() . . . . .	268
9.66.3.4 getCurrentCountrate() . . . . .	269
9.66.3.5 injectCurrentState() . . . . .	269
9.66.3.6 isAbove() . . . . .	269
9.66.3.7 isBelow() . . . . .	269



9.66.3.8 next_impl()	269
9.66.3.9 on_start()	270
9.67 Experimental::TwoStateExponentialSignalGenerator Class Reference	270
9.67.1 Constructor & Destructor Documentation	271
9.67.1.1 TwoStateExponentialSignalGenerator()	271
9.67.1.2 ~TwoStateExponentialSignalGenerator()	271
9.67.2 Member Function Documentation	271
9.67.2.1 get_next()	272
9.67.2.2 initialize()	272
9.67.2.3 on_restart()	272
9.68 Experimental::UniformSignalGenerator Class Reference	272
9.68.1 Constructor & Destructor Documentation	273
9.68.1.1 UniformSignalGenerator()	273
9.68.1.2 ~UniformSignalGenerator()	273
9.68.2 Member Function Documentation	273
9.68.2.1 get_next()	274
9.68.2.2 initialize()	274
9.68.2.3 on_restart()	274
<b>10 File Documentation</b>	<b>275</b>
10.1 Iterators.h File Reference	275
10.1.1 Macro Definition Documentation	278
10.1.1.1 BINNING_TEMPLATE_HELPER	278
10.1.2 Enumeration Type Documentation	279
10.1.2.1 CoincidenceTimestamp	279
10.1.2.2 GatedChannelInitial	279
10.1.2.3 State	279
10.2 TimeTagger.h File Reference	280
10.2.1 Macro Definition Documentation	283
10.2.1.1 channel_t	283
10.2.1.2 ErrorLog	283
10.2.1.3 ErrorLogSuppressed	283
10.2.1.4 GET_DATA_1D	284
10.2.1.5 GET_DATA_1D_OP1	284
10.2.1.6 GET_DATA_1D_OP2	284
10.2.1.7 GET_DATA_2D	284
10.2.1.8 GET_DATA_2D_OP1	285
10.2.1.9 GET_DATA_2D_OP2	285
10.2.1.10 GET_DATA_3D	285
10.2.1.11 InfoLog	285
10.2.1.12 InfoLogSuppressed	286
10.2.1.13 LogMessage	286

10.2.1.14 LogMessageSuppressed	286
10.2.1.15 timestamp_t	286
10.2.1.16 TIMETAGGER_VERSION	286
10.2.1.17 TT_API	286
10.2.1.18 WarningLog	287
10.2.1.19 WarningLogSuppressed	287
10.2.2 Typedef Documentation	287
10.2.2.1 _Iterator	287
10.2.2.2 logger_callback	287
10.2.3 Enumeration Type Documentation	287
10.2.3.1 AccessMode	287
10.2.3.2 ChannelEdge	288
10.2.3.3 FpgaLinkInterface	288
10.2.3.4 FrontendType	288
10.2.3.5 LanguageUsed	289
10.2.3.6 LogLevel	289
10.2.3.7 Resolution	289
10.2.3.8 UsageStatisticsStatus	290
10.2.4 Function Documentation	290
10.2.4.1 createTimeTagger()	290
10.2.4.2 createTimeTaggerNetwork()	290
10.2.4.3 createTimeTaggerVirtual()	291
10.2.4.4 extractDeviceLicense()	291
10.2.4.5 flashLicense()	291
10.2.4.6 freeTimeTagger()	292
10.2.4.7 getTimeTaggerChannelNumberScheme()	292
10.2.4.8 getTimeTaggerModel()	292
10.2.4.9 getTimeTaggerServerInfo()	292
10.2.4.10 getUsageStatisticsReport()	292
10.2.4.11 getUsageStatisticsStatus()	293
10.2.4.12 getVersion()	293
10.2.4.13 hasTimeTaggerVirtualLicense()	293
10.2.4.14 LogBase()	294
10.2.4.15 mergeStreamFiles()	294
10.2.4.16 scanTimeTagger()	294
10.2.4.17 scanTimeTaggerServers()	295
10.2.4.18 setCustomBitFileName()	295
10.2.4.19 setFrontend()	295
10.2.4.20 setLanguageInfo()	295
10.2.4.21 setLogger()	296
10.2.4.22 setTimeTaggerChannelNumberScheme()	296
10.2.4.23 setUsageStatisticsStatus()	297

---

10.2.5 Variable Documentation . . . . .	297
10.2.5.1 CHANNEL_UNUSED . . . . .	297
10.2.5.2 CHANNEL_UNUSED_OLD . . . . .	297
10.2.5.3 TT_CHANNEL_FALLING_EDGES . . . . .	297
10.2.5.4 TT_CHANNEL_NUMBER_SCHEME_AUTO . . . . .	297
10.2.5.5 TT_CHANNEL_NUMBER_SCHEME_ONE . . . . .	298
10.2.5.6 TT_CHANNEL_NUMBER_SCHEME_ZERO . . . . .	298
10.2.5.7 TT_CHANNEL_RISING_AND_FALLING_EDGES . . . . .	298
10.2.5.8 TT_CHANNEL_RISING_EDGES . . . . .	298



# Chapter 1

## TimeTagger

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

# Module Index

### 2.1 Modules

Here is a list of all modules:

Implementations with a Time Tagger interface . . . . .	15
All measurements and virtual channels . . . . .	16
Event counting . . . . .	17
Time histograms . . . . .	18
Fluorescence-lifetime imaging (FLIM) . . . . .	19
Time-tag-streaming . . . . .	20
Helper classes . . . . .	21
Virtual Channels . . . . .	22





## Chapter 3

# Namespace Index

### 3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

<a href="#">Experimental</a> . . . . .	<a href="#">23</a>
--	--------------------



## Chapter 4

# Hierarchical Index

### 4.1 Class Hierarchy

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

CounterData . . . . .	47
CustomLogger . . . . .	53
Event . . . . .	65
FastBinning . . . . .	72
FileReader . . . . .	73
FlimFrameInfo . . . . .	102
FrequencyStabilityData . . . . .	111
IteratorBase . . . . .	139
Coincidences . . . . .	27
Coincidence . . . . .	25
Combiner . . . . .	29
ConstantFractionDiscriminator . . . . .	32
Correlation . . . . .	35
CountBetweenMarkers . . . . .	39
Counter . . . . .	43
Countrate . . . . .	50
CustomMeasurementBase . . . . .	55
DelayedChannel . . . . .	59
Dump . . . . .	63
EventGenerator . . . . .	66
Experimental::MarkovProcessGenerator . . . . .	148
Experimental::SignalGeneratorBase . . . . .	165
Experimental::ExponentialSignalGenerator . . . . .	70
Experimental::GammaSignalGenerator . . . . .	115
Experimental::GaussianSignalGenerator . . . . .	119
Experimental::PatternSignalGenerator . . . . .	154
Experimental::PoissonSignalGenerator . . . . .	156
Experimental::TwoStateExponentialSignalGenerator . . . . .	270
Experimental::UniformSignalGenerator . . . . .	272
Experimental::SimLifetime . . . . .	170
Experimental::SimSignalSplitter . . . . .	173
Experimental::TransformCrosstalk . . . . .	257
Experimental::TransformDeadtime . . . . .	259
Experimental::TransformEfficiency . . . . .	261
Experimental::TransformGaussianBroadening . . . . .	263

FileWriter . . . . .	77
FlimAbstract . . . . .	91
Flim . . . . .	81
FlimBase . . . . .	99
FrequencyMultiplier . . . . .	105
FrequencyStability . . . . .	108
GatedChannel . . . . .	117
Histogram . . . . .	122
Histogram2D . . . . .	125
HistogramLogBins . . . . .	129
HistogramND . . . . .	133
Iterator . . . . .	136
Sampler . . . . .	158
Scope . . . . .	162
StartStop . . . . .	177
SyntheticSingleTag . . . . .	183
TimeDifferences . . . . .	188
TimeDifferencesND . . . . .	193
TimeTagStream . . . . .	252
TriggerOnCountrate . . . . .	265
OrderedBarrier . . . . .	151
OrderedPipeline . . . . .	152
OrderedBarrier::OrderInstance . . . . .	153
Experimental::SimDetector . . . . .	169
SoftwareClockState . . . . .	175
SynchronizedMeasurements . . . . .	180
Tag . . . . .	186
TimeDifferencesImpl< T > . . . . .	193
TimeTaggerBase . . . . .	220
TimeTagger . . . . .	197
TimeTaggerNetwork . . . . .	233
TimeTaggerVirtual . . . . .	248
TimeTagStreamBuffer . . . . .	254

## Chapter 5

# Class Index

### 5.1 Class List

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

<a href="#">Coincidence</a>	
<a href="#">Coincidence</a> monitor for one channel group . . . . .	25
<a href="#">Coincidences</a>	
<a href="#">Coincidence</a> monitor for many channel groups . . . . .	27
<a href="#">Combiner</a>	
Combine some channels in a virtual channel which has a tick for each tick in the input channels	29
<a href="#">ConstantFractionDiscriminator</a>	
Virtual CFD implementation which returns the mean time between a raising and a falling pair of edges . . . . .	32
<a href="#">Correlation</a>	
Auto- and Cross-correlation measurement . . . . .	35
<a href="#">CountBetweenMarkers</a>	
Simple counter where external marker signals determine the bins . . . . .	39
<a href="#">Counter</a>	
Simple counter on one or more channels . . . . .	43
<a href="#">CounterData</a>	
Helper object as return value for <a href="#">Counter::getDataObject</a> . . . . .	47
<a href="#">Countrate</a>	
Count rate on one or more channels . . . . .	50
<a href="#">CustomLogger</a>	
Helper class for setLogger . . . . .	53
<a href="#">CustomMeasurementBase</a>	
Helper class for custom measurements in Python and C# . . . . .	55
<a href="#">DelayedChannel</a>	
Simple delayed queue . . . . .	59
<a href="#">Dump</a>	
<a href="#">Dump</a> all time tags to a file . . . . .	63
<a href="#">Event</a>	
Object for the return value of <a href="#">Scope::getData</a> . . . . .	65
<a href="#">EventGenerator</a>	
Generate predefined events in a virtual channel relative to a trigger event . . . . .	66
<a href="#">Experimental::ExponentialSignalGenerator</a> . . . . .	70
<a href="#">FastBinning</a>	
Helper class for fast division with a constant divisor . . . . .	72
<a href="#">FileReader</a>	
Reads tags from the disk files, which has been created by <a href="#">FileWriter</a> . . . . .	73

<a href="#">FileWriter</a>	Compresses and stores all time tags to a file . . . . .	77
<a href="#">Flim</a>	Fluorescence lifetime imaging . . . . .	81
<a href="#">FlimAbstract</a>	Interface for FLIM measurements, <a href="#">Flim</a> and <a href="#">FlimBase</a> classes inherit from it . . . . .	91
<a href="#">FlimBase</a>	Basic measurement, containing a minimal set of features for efficiency purposes . . . . .	99
<a href="#">FlimFrameInfo</a>	Object for storing the state of <a href="#">Flim::getCurrentFrameEx</a> . . . . .	102
<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 . . . . .	105
<a href="#">FrequencyStability</a>	Allan deviation (and related metrics) calculator . . . . .	108
<a href="#">FrequencyStabilityData</a>	Return data object for <a href="#">FrequencyStability::getData</a> . . . . .	111
<a href="#">Experimental::GammaSignalGenerator</a>	. . . . .	115
<a href="#">GatedChannel</a>	An input channel is gated by a gate channel . . . . .	117
<a href="#">Experimental::GaussianSignalGenerator</a>	. . . . .	119
<a href="#">Histogram</a>	Accumulate time differences into a histogram . . . . .	122
<a href="#">Histogram2D</a>	A 2-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy . . . . .	125
<a href="#">HistogramLogBins</a>	Accumulate time differences into a histogram with logarithmic increasing bin sizes . . . . .	129
<a href="#">HistogramND</a>	A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy . . . . .	133
<a href="#">Iterator</a>	Deprecated simple event queue . . . . .	136
<a href="#">IteratorBase</a>	Base class for all iterators . . . . .	139
<a href="#">Experimental::MarkovProcessGenerator</a>	. . . . .	148
<a href="#">OrderedBarrier</a>	Helper for implementing parallel measurements . . . . .	151
<a href="#">OrderedPipeline</a>	Helper for implementing parallel measurements . . . . .	152
<a href="#">OrderedBarrier::OrderInstance</a>	Internal object for serialization . . . . .	153
<a href="#">Experimental::PatternSignalGenerator</a>	. . . . .	154
<a href="#">Experimental::PoissonSignalGenerator</a>	. . . . .	156
<a href="#">Sampler</a>	Triggered sampling measurement . . . . .	158
<a href="#">Scope</a>	<a href="#">Scope</a> measurement . . . . .	162
<a href="#">Experimental::SignalGeneratorBase</a>	. . . . .	165
<a href="#">Experimental::SimDetector</a>	. . . . .	169
<a href="#">Experimental::SimLifetime</a>	. . . . .	170
<a href="#">Experimental::SimSignalSplitter</a>	. . . . .	173
<a href="#">SoftwareClockState</a>	. . . . .	175
<a href="#">StartStop</a>	Simple start-stop measurement . . . . .	177
<a href="#">SynchronizedMeasurements</a>	Start, stop and clear several measurements synchronized . . . . .	180

<a href="#">SyntheticSingleTag</a>	
Synthetic trigger timetag generator . . . . .	183
<a href="#">Tag</a>	
Single event on a channel . . . . .	186
<a href="#">TimeDifferences</a>	
Accumulates the time differences between clicks on two channels in one or more histograms . . . . .	188
<a href="#">TimeDifferencesImpl&lt; T &gt;</a> . . . . .	193
<a href="#">TimeDifferencesND</a>	
Accumulates the time differences between clicks on two channels in a multi-dimensional histogram . . . . .	193
<a href="#">TimeTagger</a>	
Backend for the <a href="#">TimeTagger</a> . . . . .	197
<a href="#">TimeTaggerBase</a>	
Basis interface for all Time Tagger classes . . . . .	220
<a href="#">TimeTaggerNetwork</a>	
Network <a href="#">TimeTagger</a> client . . . . .	233
<a href="#">TimeTaggerVirtual</a>	
Virtual <a href="#">TimeTagger</a> based on dump files . . . . .	248
<a href="#">TimeTagStream</a>	
Access the time tag stream . . . . .	252
<a href="#">TimeTagStreamBuffer</a>	
Return object for <a href="#">TimeTagStream::getData</a> . . . . .	254
<a href="#">Experimental::TransformCrosstalk</a> . . . . .	257
<a href="#">Experimental::TransformDeadtime</a> . . . . .	259
<a href="#">Experimental::TransformEfficiency</a> . . . . .	261
<a href="#">Experimental::TransformGaussianBroadening</a> . . . . .	263
<a href="#">TriggerOnCountrate</a>	
Inject trigger events when exceeding or falling below a given count rate within a rolling time window . . . . .	265
<a href="#">Experimental::TwoStateExponentialSignalGenerator</a> . . . . .	270
<a href="#">Experimental::UniformSignalGenerator</a> . . . . .	272





## Chapter 6

# File Index

### 6.1 File List

Here is a list of all files with brief descriptions:

<a href="#">Iterators.h</a> . . . . .	275
<a href="#">TimeTagger.h</a> . . . . .	280



## Chapter 7

# Module Documentation

### 7.1 Implementations with a Time Tagger interface

#### Classes

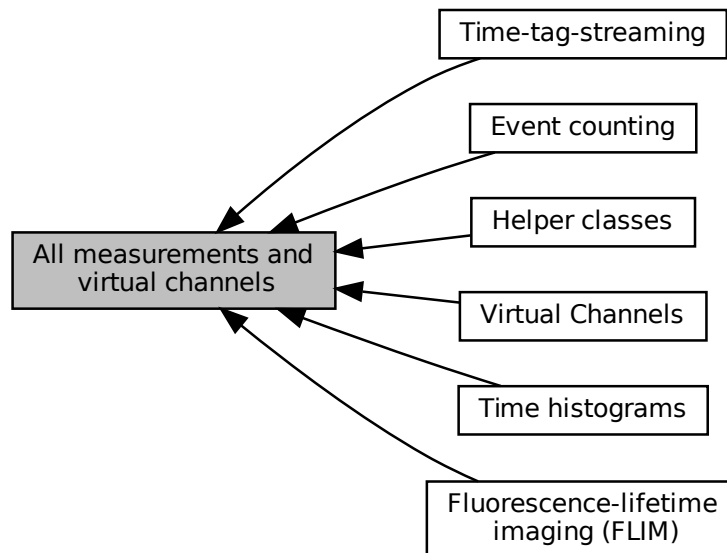
- class [TimeTaggerBase](#)  
*Basis interface for all Time Tagger classes.*
- class [TimeTaggerVirtual](#)  
*virtual [TimeTagger](#) based on dump files*
- class [TimeTagger](#)  
*backend for the [TimeTagger](#).*

#### 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:



### Modules

- [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.*

- [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](#)

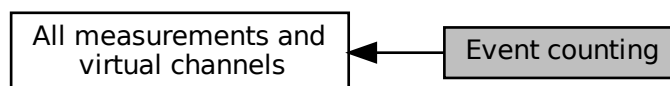
*A N-dimensional histogram of time differences. This can be used in measurements similar to 2D NRM spectroscopy.*

#### 7.2.1 Detailed Description

Base iterators for photon counting applications.

## 7.3 Event counting

Collaboration diagram for Event counting:



### Classes

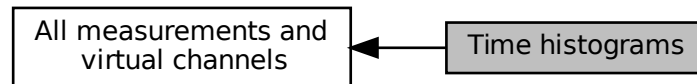
- class [CountBetweenMarkers](#)  
*a simple counter where external marker signals determine the bins*
- class [Counter](#)  
*a simple counter on one or more channels*
- class [Countrate](#)  
*count rate on one or more channels*

### 7.3.1 Detailed Description

## 7.4 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.*

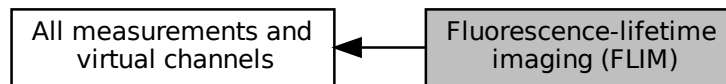
### 7.4.1 Detailed Description

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

## 7.5 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.*

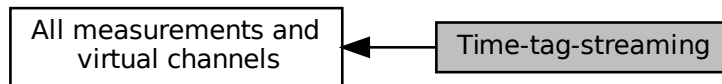
### 7.5.1 Detailed Description

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

## 7.6 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*

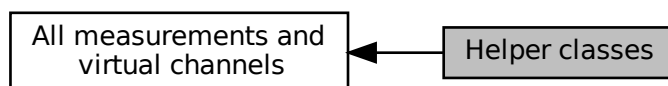
### 7.6.1 Detailed Description

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



## 7.7 Helper classes

Collaboration diagram for Helper classes:



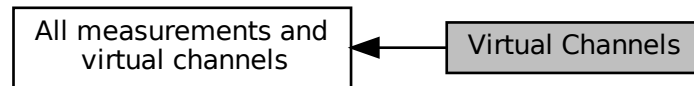
### Classes

- 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.*
- class [FrequencyStability](#)  
*Allan deviation (and related metrics) calculator.*

#### 7.7.1 Detailed Description

## 7.8 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 raising and a falling pair of edges*
- class [EventGenerator](#)  
*Generate predefined events in a virtual channel relative to a trigger event.*

### 7.8.1 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

#### Classes

- class [ExponentialSignalGenerator](#)
- class [GammaSignalGenerator](#)
- class [GaussianSignalGenerator](#)
- class [MarkovProcessGenerator](#)
- class [PatternSignalGenerator](#)
- class [PoissonSignalGenerator](#)
- class [SignalGeneratorBase](#)
- class [SimDetector](#)
- class [SimLifetime](#)
- class [SimSignalSplitter](#)
- class [TransformCrosstalk](#)
- class [TransformDeadtime](#)
- class [TransformEfficiency](#)
- class [TransformGaussianBroadening](#)
- class [TwoStateExponentialSignalGenerator](#)
- class [UniformSignalGenerator](#)



## Chapter 9

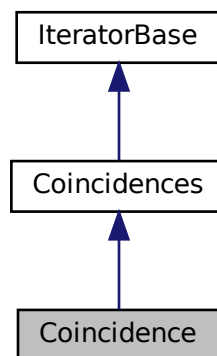
# Class Documentation

### 9.1 Coincidence Class Reference

a coincidence monitor for one channel group

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

Inheritance diagram for Coincidence:



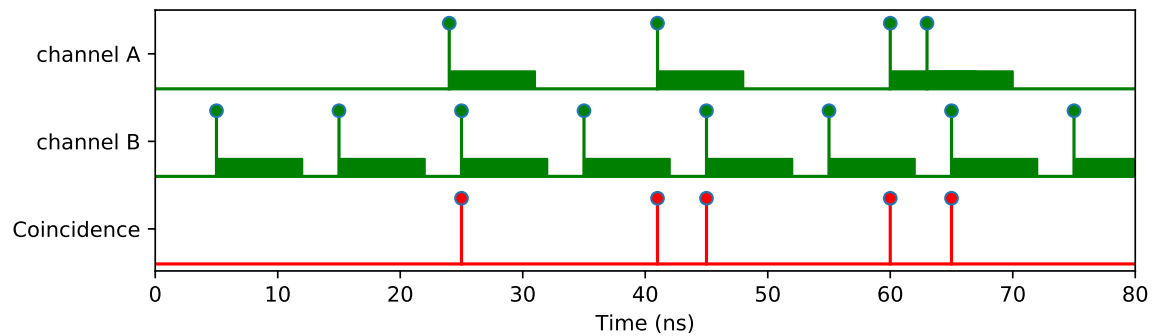
#### Public Member Functions

- `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*

## Additional Inherited Members

### 9.1.1 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](#).

### 9.1.2 Constructor & Destructor Documentation

#### 9.1.2.1 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)

### 9.1.3 Member Function Documentation

### 9.1.3.1 getChannel()

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

virtual channel which contains the coincidences

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

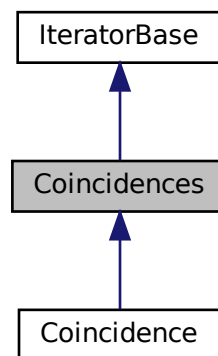
- [Iterators.h](#)

## 9.2 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)

### Protected Member Functions

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

## Additional Inherited Members

### 9.2.1 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 coincidenceWindow [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.

### 9.2.2 Constructor & Destructor Documentation

#### 9.2.2.1 Coincidences()

```
Coincidences::Coincidences (
    TimeTaggerBase * tagger,
    std::vector< std::vector< channel_t >> coincidenceGroups,
    timestamp_t coincidenceWindow,
    CoincidenceTimestamp timestamp = CoincidenceTimestamp::Last )
```

construct a [Coincidences](#)

##### Parameters

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

#### 9.2.2.2 ~Coincidences()

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

### 9.2.3 Member Function Documentation

#### 9.2.3.1 getChannels()

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

fetches the block of virtual channels for those coincidence groups



### 9.2.3.2 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](#).

### 9.2.3.3 setCoincidenceWindow()

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

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

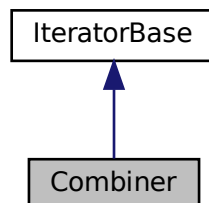
- [Iterators.h](#)

## 9.3 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*

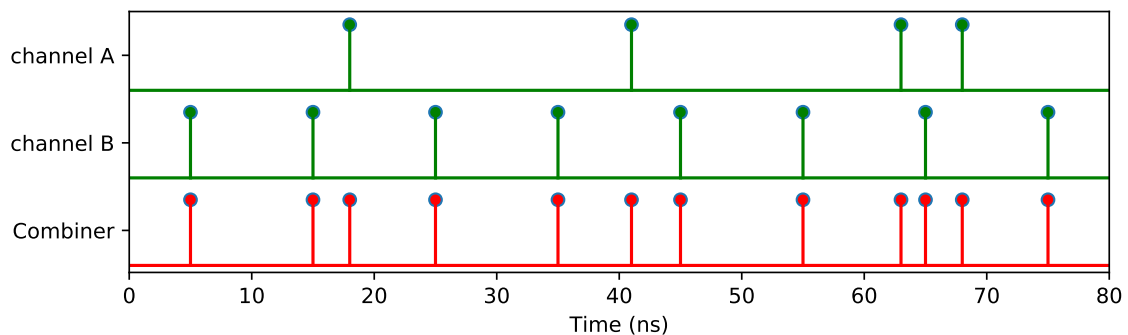
## Protected Member Functions

- `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.*

## Additional Inherited Members

### 9.3.1 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.

### 9.3.2 Constructor & Destructor Documentation

#### 9.3.2.1 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

**9.3.2.2 ~Combiner()**

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

**9.3.3 Member Function Documentation****9.3.3.1 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](#).

**9.3.3.2 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.

**9.3.3.3 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.

#### 9.3.3.4 getData()

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

get sum of counts

deprecated, use getChannelCounts instead.

#### 9.3.3.5 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:

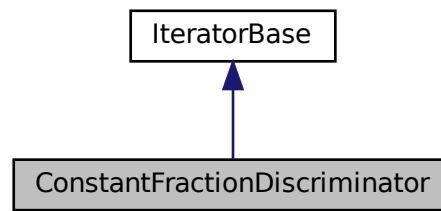
- [Iterators.h](#)

## 9.4 ConstantFractionDiscriminator Class Reference

a virtual CFD implementation which returns the mean time between a raising 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*

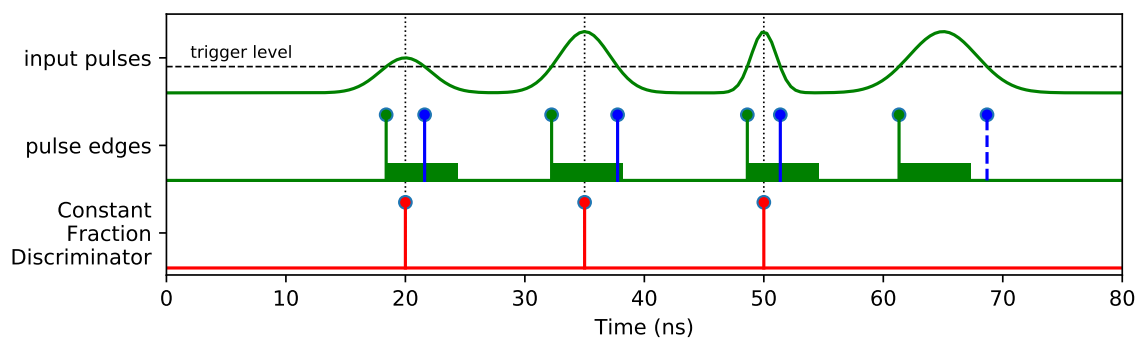
## Protected Member Functions

- [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*

## Additional Inherited Members

### 9.4.1 Detailed Description

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



## 9.4.2 Constructor & Destructor Documentation

### 9.4.2.1 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 raising+falling pairs must be given
<i>search_window</i>	interval for the CFD window, must be positive

### 9.4.2.2 ~ConstantFractionDiscriminator()

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

## 9.4.3 Member Function Documentation

### 9.4.3.1 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.

### 9.4.3.2 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](#).

**9.4.3.3 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:

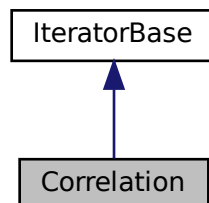
- [Iterators.h](#)

**9.5 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< long long \*(size\_t)> array\_out)  
*returns a vector of size n\_bins containing the time bins in ps*

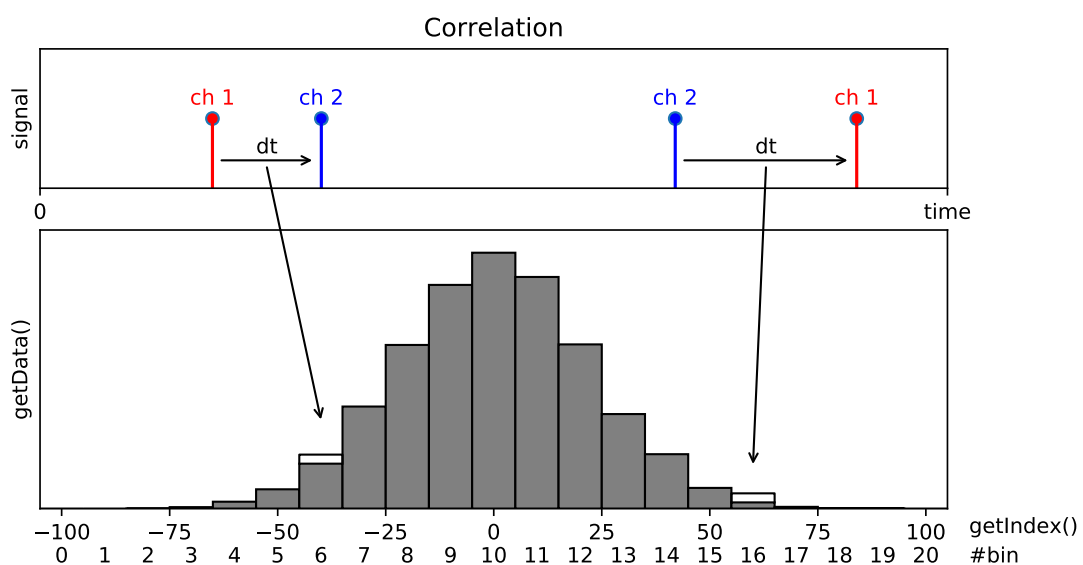
## Protected Member Functions

- 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.*

## Additional Inherited Members

### 9.5.1 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.



## 9.5.2 Constructor & Destructor Documentation

### 9.5.2.1 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

### 9.5.2.2 ~Correlation()

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

destructor of the [Correlation](#) measurement

## 9.5.3 Member Function Documentation

### 9.5.3.1 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](#).

### 9.5.3.2 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

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

### 9.5.3.3 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

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

### 9.5.3.4 getIndex()

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

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

#### Parameters

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

### 9.5.3.5 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

<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:

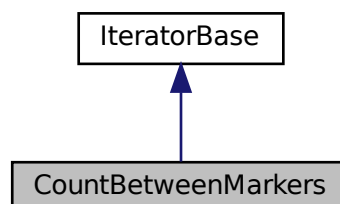
- [Iterators.h](#)

## 9.6 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< long long *(size_t)> array_out`)  
*fetches the widths of each bins*
- `void getIndex` (`std::function< long long *(size_t)> array_out`)  
*fetches the starting time of each bin*

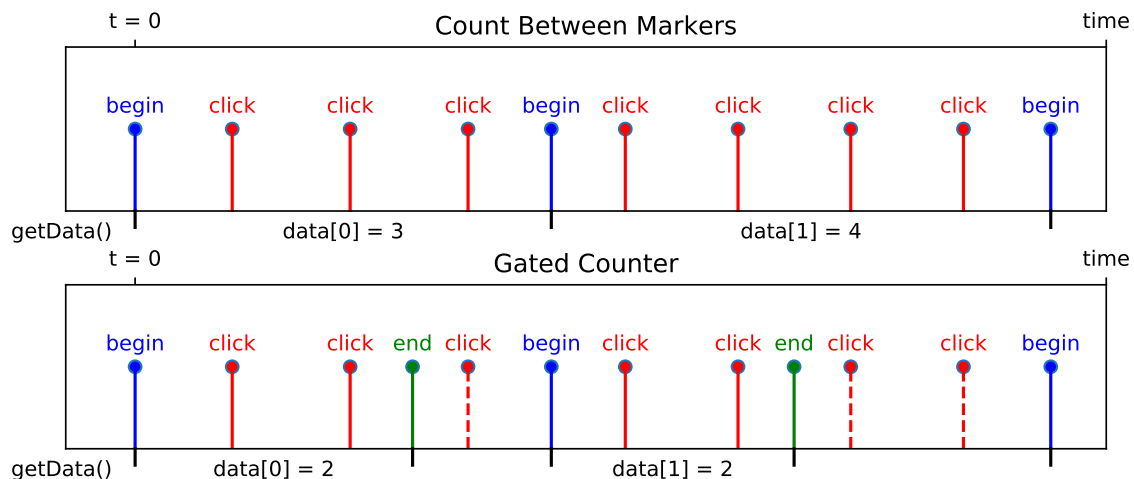
## Protected Member Functions

- `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.*

## Additional Inherited Members

### 9.6.1 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.

## 9.6.2 Constructor & Destructor Documentation

### 9.6.2.1 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

### 9.6.2.2 ~CountBetweenMarkers()

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

## 9.6.3 Member Function Documentation

### 9.6.3.1 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](#).

### 9.6.3.2 getBinWidths()

```
void CountBetweenMarkers::getBinWidths (
    std::function< long long *(size_t)> array_out )
```

fetches the widths of each bins

### 9.6.3.3 getData()

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

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

### 9.6.3.4 getIndex()

```
void CountBetweenMarkers::getIndex (
    std::function< long long *(size_t)> array_out )
```

fetches the starting time of each bin

### 9.6.3.5 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](#).

### 9.6.3.6 ready()

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

Returns true when the entire array is filled.

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

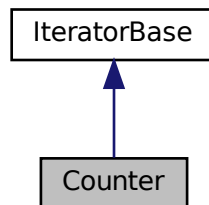
- [Iterators.h](#)

## 9.7 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< long long \*(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.*

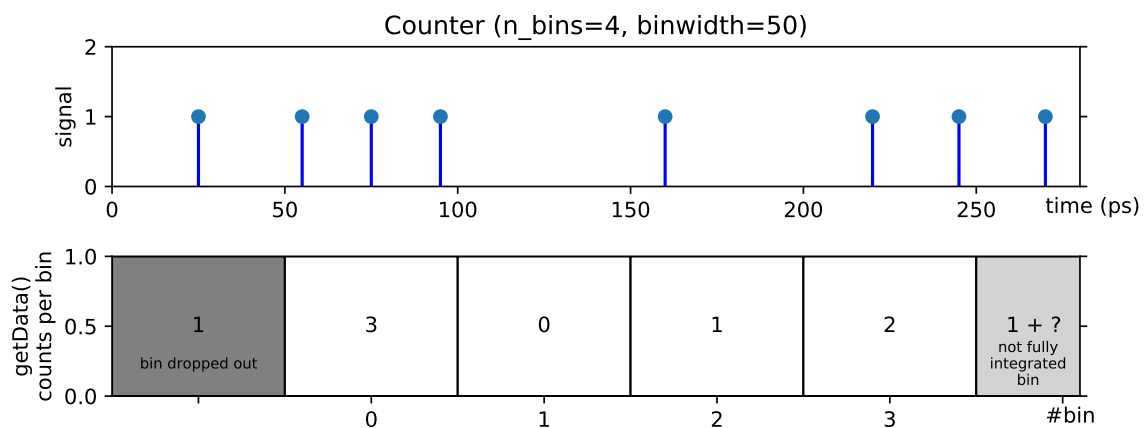
## Protected Member Functions

- 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*

## Additional Inherited Members

### 9.7.1 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.

### 9.7.2 Constructor & Destructor Documentation

#### 9.7.2.1 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)

## 9.7.2.2 ~Counter()

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

## 9.7.3 Member Function Documentation

## 9.7.3.1 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](#).

## 9.7.3.2 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

### 9.7.3.3 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

### 9.7.3.4 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

### 9.7.3.5 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

### 9.7.3.6 getIndex()

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

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

### 9.7.3.7 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](#).

### 9.7.3.8 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](#)

## 9.8 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 per bin and per channel*
- 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 rececnt bin*
- void [getIndex](#) (std::function< long long \*(size\_t)> array\_out)  
*get an index which corresponds to the timestamp of these bins*
- void [getTime](#) (std::function< long long \*(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< int \*(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*

### 9.8.1 Detailed Description

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

This object stores the result of up to n\_values bins.

### 9.8.2 Constructor & Destructor Documentation

#### 9.8.2.1 ~CounterData()

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

### 9.8.3 Member Function Documentation

### 9.8.3.1 getChannels()

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

get the configured list of channels

### 9.8.3.2 getData()

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

get the amount of clicks per bin and per channel

### 9.8.3.3 getDataNormalized()

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

get the average rate of clicks per bin and per channel

### 9.8.3.4 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

### 9.8.3.5 getIndex()

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

get an index which corresponds to the timestamp of these bins

### 9.8.3.6 getOverflowMask()

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

get if the bins were in overflow

### 9.8.3.7 getTime()

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

get the timestamp of the bins since the last clear

## 9.8.4 Member Data Documentation

### 9.8.4.1 dropped\_bins

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

number of bins which have been dropped because n\_bins has been exceeded, usually 0

### 9.8.4.2 overflow

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

has anything been in overflow mode

### 9.8.4.3 size

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

number of returned bins

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

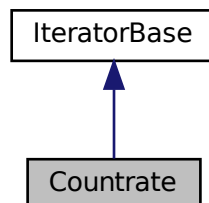
- [Iterators.h](#)

## 9.9 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*

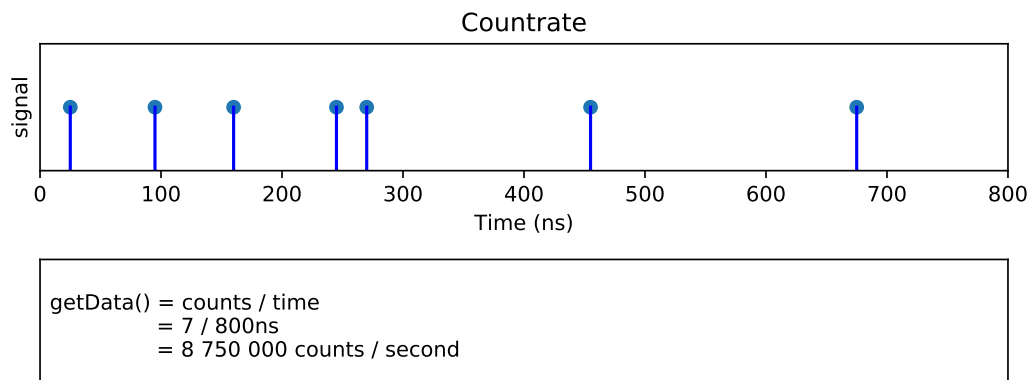
## Protected Member Functions

- `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*

## Additional Inherited Members

### 9.9.1 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.

### 9.9.2 Constructor & Destructor Documentation

#### 9.9.2.1 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

**9.9.2.2 ~Countrate()**

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

**9.9.3 Member Function Documentation****9.9.3.1 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](#).

**9.9.3.2 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.

**9.9.3.3 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.

**9.9.3.4 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](#).

**9.9.3.5 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](#)

## 9.10 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

### 9.10.1 Detailed Description

Helper class for setLogger.

## 9.10.2 Constructor & Destructor Documentation

### 9.10.2.1 CustomLogger()

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

### 9.10.2.2 ~CustomLogger()

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

## 9.10.3 Member Function Documentation

### 9.10.3.1 disable()

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

### 9.10.3.2 enable()

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

### 9.10.3.3 Log()

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

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

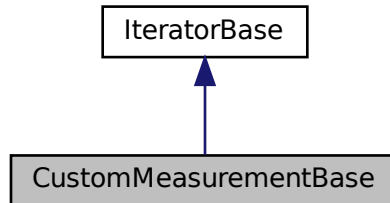
- [TimeTagger.h](#)

## 9.11 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](#) ()

### 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*

## Additional Inherited Members

### 9.11.1 Detailed Description

Helper class for custom measurements in Python and C#.

### 9.11.2 Constructor & Destructor Documentation

#### 9.11.2.1 CustomMeasurementBase()

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

#### 9.11.2.2 ~CustomMeasurementBase()

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

### 9.11.3 Member Function Documentation

#### 9.11.3.1 \_lock()

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

#### 9.11.3.2 \_unlock()

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

#### 9.11.3.3 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](#).

#### 9.11.3.4 finalize\_init()

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

#### 9.11.3.5 is\_running()

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

#### 9.11.3.6 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](#).

#### 9.11.3.7 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]
```

#### 9.11.3.8 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](#).

#### 9.11.3.9 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](#).

#### 9.11.3.10 register\_channel()

```
void CustomMeasurementBase::register_channel (
    channel\_t channel )
```

#### 9.11.3.11 stop\_all\_custom\_measurements()

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

#### 9.11.3.12 unregister\_channel()

```
void CustomMeasurementBase::unregister_channel (
    channel\_t channel )
```

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

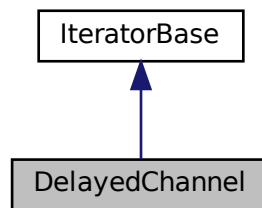
- [Iterators.h](#)

## 9.12 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.*

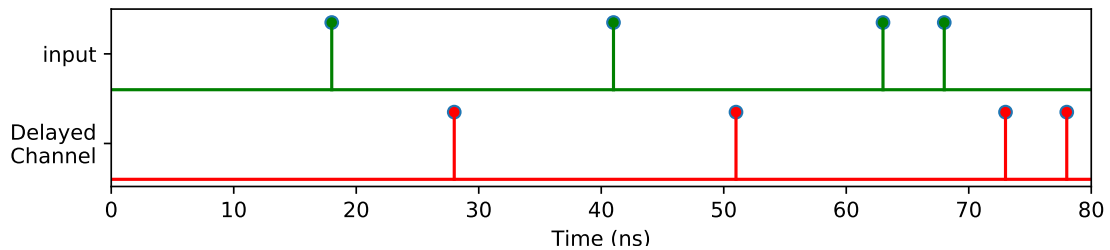
### Protected Member Functions

- `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*

## Additional Inherited Members

### 9.12.1 Detailed Description

a simple delayed queue



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

### 9.12.2 Constructor & Destructor Documentation

#### 9.12.2.1 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

#### 9.12.2.2 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

**9.12.2.3 ~DelayedChannel()**

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

**9.12.3 Member Function Documentation****9.12.3.1 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.

**9.12.3.2 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.

**9.12.3.3 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](#).

**9.12.3.4 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](#).

**9.12.3.5 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:

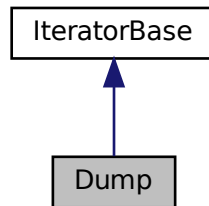
- [Iterators.h](#)

## 9.13 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` ()

### 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*

### Additional Inherited Members

#### 9.13.1 Detailed Description

dump all time tags to a file

## 9.13.2 Constructor & Destructor Documentation

### 9.13.2.1 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)

### 9.13.2.2 ~Dump()

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

## 9.13.3 Member Function Documentation

### 9.13.3.1 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](#).

### 9.13.3.2 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](#).

**9.13.3.3 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](#).

**9.13.3.4 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](#)

**9.14 Event Struct Reference**

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

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

## Public Attributes

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

### 9.14.1 Detailed Description

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

### 9.14.2 Member Data Documentation

#### 9.14.2.1 state

[State](#) Event::state

#### 9.14.2.2 time

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

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

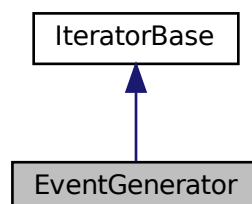
- [Iterators.h](#)

## 9.15 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*

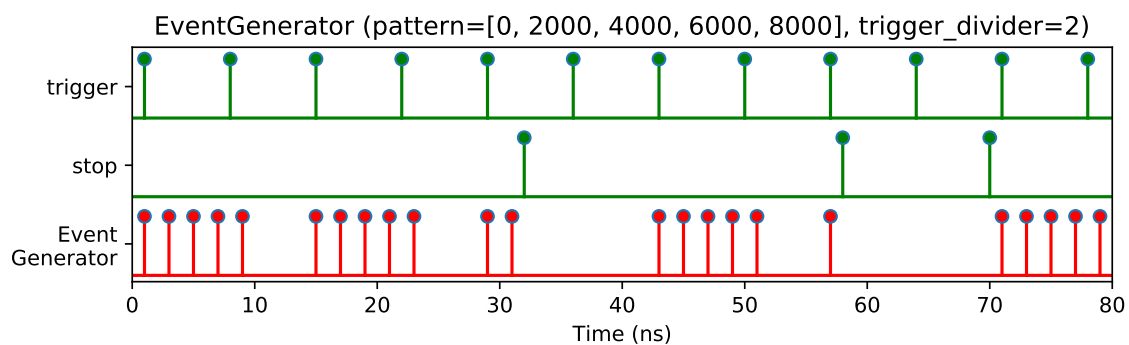
## Protected Member Functions

- [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*

## Additional Inherited Members

### 9.15.1 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 trigger\_offset 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.

### 9.15.2 Constructor & Destructor Documentation

### 9.15.2.1 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

### 9.15.2.2 ~EventGenerator()

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

## 9.15.3 Member Function Documentation

### 9.15.3.1 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](#).

### 9.15.3.2 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.



### 9.15.3.3 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](#).

### 9.15.3.4 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](#).

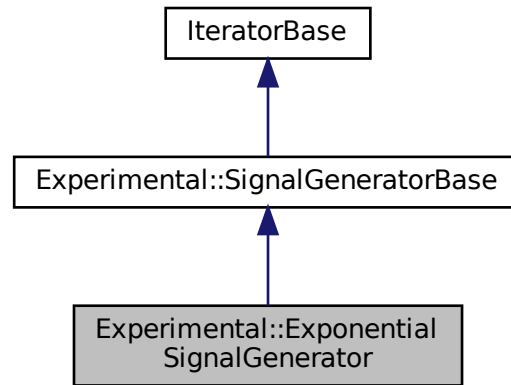
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.16 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](#) ()

### Protected Member Functions

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

### Additional Inherited Members

#### 9.16.1 Constructor & Destructor Documentation

##### 9.16.1.1 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.

## 9.16.1.2 ~ExponentialSignalGenerator()

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

## 9.16.2 Member Function Documentation

## 9.16.2.1 get\_next()

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

Implements [Experimental::SignalGeneratorBase](#).

## 9.16.2.2 initialize()

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

Implements [Experimental::SignalGeneratorBase](#).

## 9.16.2.3 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](#)

## 9.17 FastBinning Class Reference

Helper class for fast division with a constant divisor.

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

### Public Types

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

#### 9.17.1 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.

#### 9.17.2 Member Enumeration Documentation

##### 9.17.2.1 Mode

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

##### Enumerator

ConstZero	
Dividend	
PowerOfTwo	
FixedPoint_32	
FixedPoint_64	
Divide_32	
Divide_64	

### 9.17.3 Constructor & Destructor Documentation

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

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

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

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

### 9.17.4 Member Function Documentation

#### 9.17.4.1 divide()

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

#### 9.17.4.2 getMode()

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

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

- [Iterators.h](#)

## 9.18 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.*

### 9.18.1 Detailed Description

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

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

### 9.18.2 Constructor & Destructor Documentation

#### 9.18.2.1 [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
------------------	---

### 9.18.2.2 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
-----------------	--

### 9.18.2.3 ~FileReader()

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

## 9.18.3 Member Function Documentation

### 9.18.3.1 getChannelList()

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

Fetches channels from the input file.

#### Returns

a vector of channels from the input file.

### 9.18.3.2 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.

### 9.18.3.3 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

**9.18.3.4 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

**9.18.3.5 getLastMarker()**

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

return the last processed marker from the file.

**Returns**

the last marker from the file

**9.18.3.6 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](#)

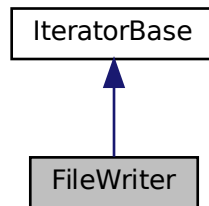


## 9.19 FileWriter Class Reference

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.*

### 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*

## Additional Inherited Members

### 9.19.1 Detailed Description

compresses and stores all time tags to a file

### 9.19.2 Constructor & Destructor Documentation

#### 9.19.2.1 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

#### 9.19.2.2 ~FileWriter()

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

### 9.19.3 Member Function Documentation

#### 9.19.3.1 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](#).

### 9.19.3.2 getMaxFileSize()

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

fetches the maximum file size. Please see `setMaxFileSize` for more details.

#### Returns

the maximum file size in bytes

### 9.19.3.3 getTotalEvents()

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

queries the total amount of events stored in all files

#### Returns

the total amount of events stored

### 9.19.3.4 getTotalSize()

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

queries the total amount of bytes stored in all files

#### Returns

the total amount of bytes stored

### 9.19.3.5 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](#).

**9.19.3.6 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](#).

**9.19.3.7 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](#).

**9.19.3.8 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
---------------	-----------------------------------

### 9.19.3.9 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
----------------------	--------------------------------

### 9.19.3.10 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:

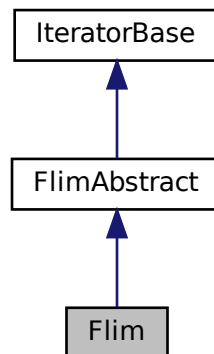
- [Iterators.h](#)

## 9.20 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](#) ()
- [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](#)< [long long](#) \*([size\\_t](#))> array\_out)  
*a vector of size n\_bins containing the time bins in ps*

## Protected Member Functions

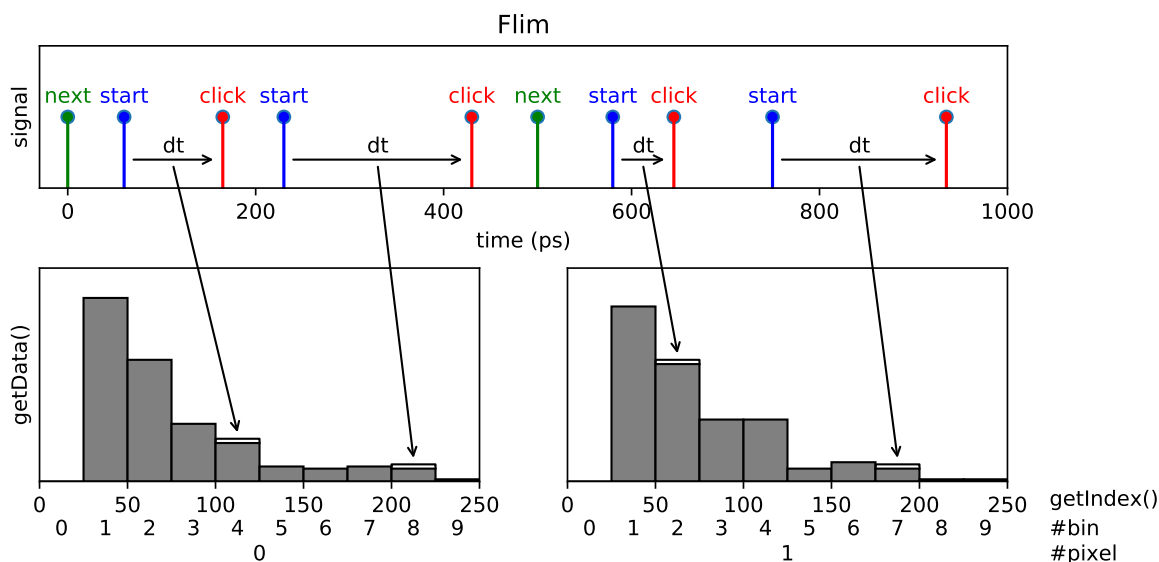
- void `on_frame_end()` override
- void `clear_impl()` override  
clear *Iterator* state.
- uint32\_t `get_ready_index` (int32\_t index)
- virtual void `frameReady` (uint32\_t frame\_number, std::vector< uint32\_t > &data, std::vector< timestamp\_t > &pixel\_begin\_times, std::vector< timestamp\_t > &pixel\_end\_times, timestamp\_t frame\_begin\_time, timestamp\_t frame\_end\_time)

## Protected 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`

### 9.20.1 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.

## 9.20.2 Constructor & Destructor Documentation

### 9.20.2.1 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)
<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.

### 9.20.2.2 ~Flim()

```

Flim::~Flim ( )

```

## 9.20.3 Member Function Documentation



### 9.20.3.1 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 [FlimAbstract](#).

### 9.20.3.2 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]
```

### 9.20.3.3 get\_ready\_index()

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

### 9.20.3.4 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

### 9.20.3.5 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

### 9.20.3.6 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.

### 9.20.3.7 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.

### 9.20.3.8 getIndex()

```
void Flim::getIndex (
    std::function< long long *(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.

### 9.20.3.9 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

**9.20.3.10 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  $\geq$  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
--------------	---

**9.20.3.11 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  $\geq$  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

**9.20.3.12 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.

**9.20.3.13 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 far.

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.

**9.20.3.14 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.

### 9.20.3.15 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.

### 9.20.3.16 on\_frame\_end()

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

Implements [FlimAbstract](#).

## 9.20.4 Member Data Documentation

### 9.20.4.1 accum\_diffs

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

### 9.20.4.2 back\_frames

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

### 9.20.4.3 captured\_frames

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

### 9.20.4.4 frame\_begins

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

#### 9.20.4.5 frame\_ends

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

#### 9.20.4.6 last\_frame

`int32_t Flim::last_frame` [protected]

#### 9.20.4.7 pixels\_completed

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

#### 9.20.4.8 summed\_frames

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

#### 9.20.4.9 swap\_chain\_lock

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

#### 9.20.4.10 total\_frames

`uint32_t Flim::total_frames` [protected]

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

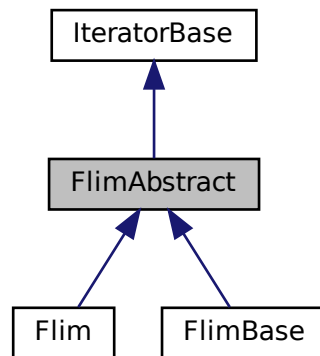
- [Iterators.h](#)

## 9.21 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*

### Protected Member Functions

- [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*
- [virtual void](#) [on\\_frame\\_end](#) ()=0

## 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](#)

### 9.21.1 Detailed Description

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

### 9.21.2 Constructor & Destructor Documentation

#### 9.21.2.1 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
<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.

## 9.21.2.2 ~FlimAbstract()

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

## 9.21.3 Member Function Documentation

## 9.21.3.1 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](#).

Reimplemented in [Flim](#).

### 9.21.3.2 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`.

### 9.21.3.3 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](#).

### 9.21.3.4 on\_frame\_end()

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

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

### 9.21.3.5 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](#).

### 9.21.3.6 process\_tags()

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

## 9.21.4 Member Data Documentation

### 9.21.4.1 acquiring

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

### 9.21.4.2 acquisition\_lock

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

### 9.21.4.3 binner

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

### 9.21.4.4 binwidth

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

#### 9.21.4.5 click\_channel

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

#### 9.21.4.6 current\_frame\_begin

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

#### 9.21.4.7 current\_frame\_end

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

#### 9.21.4.8 data\_base

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

#### 9.21.4.9 finish\_after\_outputframe

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

#### 9.21.4.10 frame

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

#### 9.21.4.11 frame\_acquisition

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

#### 9.21.4.12 frame\_begin\_channel

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

#### 9.21.4.13 frames\_completed

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

#### 9.21.4.14 initialized

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

#### 9.21.4.15 n\_bins

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

#### 9.21.4.16 n\_frame\_average

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

#### 9.21.4.17 n\_pixels

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

#### 9.21.4.18 pixel\_acquisition

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

#### 9.21.4.19 pixel\_begin\_channel

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

#### 9.21.4.20 pixel\_begins

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

#### 9.21.4.21 pixel\_end\_channel

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

#### 9.21.4.22 pixel\_ends

```
std::vector<timestamp_t> FlimAbstract::pixel_ends [protected]
```

#### 9.21.4.23 pixels\_processed

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

#### 9.21.4.24 previous\_starts

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

#### 9.21.4.25 start\_channel

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

#### 9.21.4.26 ticks

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

#### 9.21.4.27 time\_window

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

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

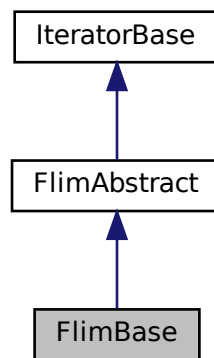
- [Iterators.h](#)

## 9.22 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*

### Protected Member Functions

- [void on\\_frame\\_end](#) () override
- [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 Attributes

- [uint32\\_t total\\_frames](#)

### 9.22.1 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.

### 9.22.2 Constructor & Destructor Documentation

#### 9.22.2.1 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.



### 9.22.2.2 ~FlimBase()

```
FlimBase::~FlimBase ( )
```

## 9.22.3 Member Function Documentation

### 9.22.3.1 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]
```

### 9.22.3.2 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.

### 9.22.3.3 on\_frame\_end()

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

Implements [FlimAbstract](#).

## 9.22.4 Member Data Documentation

### 9.22.4.1 total\_frames

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

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

- [Iterators.h](#)

## 9.23 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< [long long](#) \*([size\\_t](#))> array\_out)
- [void getPixelEnds](#) (std::function< [long long](#) \*([size\\_t](#))> array\_out)

### Public Attributes

- [uint32\\_t pixels](#)
- [uint32\\_t bins](#)
- [int32\\_t frame\\_number](#)
- [uint32\\_t pixel\\_position](#)
- [bool valid](#)

### 9.23.1 Detailed Description

object for storing the state of [Flim::getCurrentFrameEx](#)

### 9.23.2 Constructor & Destructor Documentation

#### 9.23.2.1 ~FlimFrameInfo()

```
FlimFrameInfo::~FlimFrameInfo ( )
```

### 9.23.3 Member Function Documentation

#### 9.23.3.1 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..

#### 9.23.3.2 getHistograms()

```
void FlimFrameInfo::getHistograms (
    std::function< uint32_t *(size_t, size_t)> array_out )
```

#### 9.23.3.3 getIntensities()

```
void FlimFrameInfo::getIntensities (
    std::function< float *(size_t)> array_out )
```

#### 9.23.3.4 getPixelBegins()

```
void FlimFrameInfo::getPixelBegins (
    std::function< long long *(size_t)> array_out )
```

#### 9.23.3.5 getPixelEnds()

```
void FlimFrameInfo::getPixelEnds (
    std::function< long long *(size_t)> array_out )
```

#### 9.23.3.6 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.

#### 9.23.3.7 getSummedCounts()

```
void FlimFrameInfo::getSummedCounts (
    std::function< uint64_t *(size_t)> array_out )
```

#### 9.23.3.8 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.

### 9.23.4 Member Data Documentation

#### 9.23.4.1 bins

```
uint32_t FlimFrameInfo::bins
```

#### 9.23.4.2 frame\_number

```
int32_t FlimFrameInfo::frame_number
```

#### 9.23.4.3 pixel\_position

```
uint32_t FlimFrameInfo::pixel_position
```

#### 9.23.4.4 pixels

```
uint32_t FlimFrameInfo::pixels
```

### 9.23.4.5 valid

```
bool FlimFrameInfo::valid
```

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

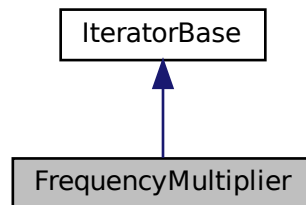
- [Iterators.h](#)

## 9.24 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 ()

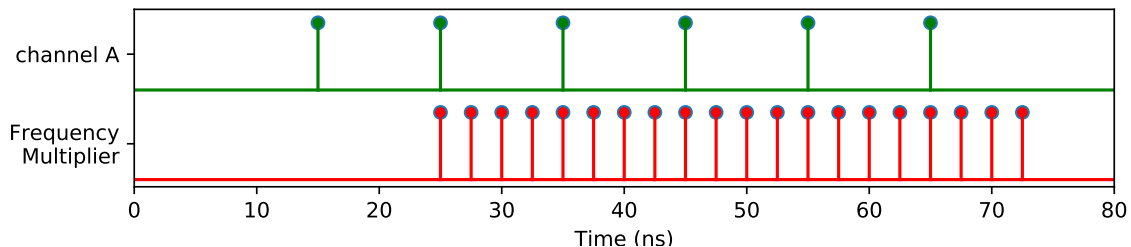
### Protected Member Functions

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

## Additional Inherited Members

### 9.24.1 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 reduced 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.

### 9.24.2 Constructor & Destructor Documentation

#### 9.24.2.1 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

### 9.24.2.2 ~FrequencyMultiplier()

```
FrequencyMultiplier::~~FrequencyMultiplier ( )
```

## 9.24.3 Member Function Documentation

### 9.24.3.1 getChannel()

```
channel_t FrequencyMultiplier::getChannel ( )
```

### 9.24.3.2 getMultiplier()

```
int32_t FrequencyMultiplier::getMultiplier ( )
```

### 9.24.3.3 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:

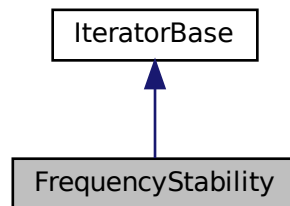
- [Iterators.h](#)

## 9.25 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*

### 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*

### Additional Inherited Members

#### 9.25.1 Detailed Description

Allan deviation (and related metrics) calculator.

It shall analyse 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 analyse 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.

## 9.25.2 Constructor & Destructor Documentation

### 9.25.2.1 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

**9.25.2.2 ~FrequencyStability()**

```
FrequencyStability::~~FrequencyStability ( )
```

**9.25.3 Member Function Documentation****9.25.3.1 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](#).

**9.25.3.2 getDataObject()**

```
FrequencyStabilityData FrequencyStability::getDataObject ( )
```

get a return object with all data in a synchronized way

**9.25.3.3 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](#).

## 9.25.3.4 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](#)

## 9.26 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*

### 9.26.1 Detailed Description

return data object for FrequencyStability::getData.

### 9.26.2 Constructor & Destructor Documentation

#### 9.26.2.1 ~FrequencyStabilityData()

```
FrequencyStabilityData::~FrequencyStabilityData ( )
```

### 9.26.3 Member Function Documentation

#### 9.26.3.1 getADEV()

```
void FrequencyStabilityData::getADEV (
    std::function< double *(size_t)> array_out )
```

returns the overlapping Allan deviation

#### 9.26.3.2 getADEVScaled()

```
void FrequencyStabilityData::getADEVScaled (
    std::function< double *(size_t)> array_out )
```

returns the scaled version of the overlapping Allan deviation

### 9.26.3.3 getHDEV()

```
void FrequencyStabilityData::getHDEV (
    std::function< double *(size_t)> array_out )
```

returns the overlapping Hadamard deviation

### 9.26.3.4 getHDEVScaled()

```
void FrequencyStabilityData::getHDEVScaled (
    std::function< double *(size_t)> array_out )
```

returns the scaled version of the overlapping Hadamard deviation

### 9.26.3.5 getMDEV()

```
void FrequencyStabilityData::getMDEV (
    std::function< double *(size_t)> array_out )
```

returns the modified overlapping Allan deviation

### 9.26.3.6 getSTDD()

```
void FrequencyStabilityData::getSTDD (
    std::function< double *(size_t)> array_out )
```

returns the standard derivation of each period pair

### 9.26.3.7 getTau()

```
void FrequencyStabilityData::getTau (
    std::function< double *(size_t)> array_out )
```

returns the analysis position of all deviations

### 9.26.3.8 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.

### 9.26.3.9 getTraceFrequency()

```
void FrequencyStabilityData::getTraceFrequency (
    std::function< double *(size_t)> array_out )
```

returns a trace of the last normalized frequency error samples in pp1

### 9.26.3.10 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.

### 9.26.3.11 getTraceIndex()

```
void FrequencyStabilityData::getTraceIndex (
    std::function< double *(size_t)> array_out )
```

returns the timestamps of the traces in seconds

### 9.26.3.12 getTracePhase()

```
void FrequencyStabilityData::getTracePhase (
    std::function< double *(size_t)> array_out )
```

returns a trace of the last phase samples in seconds

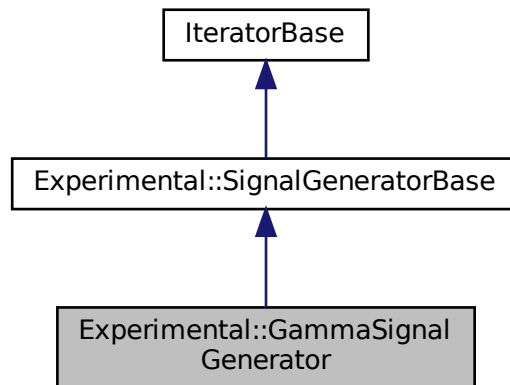
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.27 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](#) ()

### Protected Member Functions

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

### Additional Inherited Members

#### 9.27.1 Constructor & Destructor Documentation

##### 9.27.1.1 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.

**9.27.1.2 ~GammaSignalGenerator()**

```
Experimental::GammaSignalGenerator::~~GammaSignalGenerator ( )
```

**9.27.2 Member Function Documentation****9.27.2.1 get\_next()**

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

Implements [Experimental::SignalGeneratorBase](#).

**9.27.2.2 initialize()**

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

Implements [Experimental::SignalGeneratorBase](#).

**9.27.2.3 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](#)

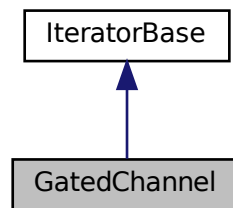


## 9.28 GatedChannel Class Reference

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*

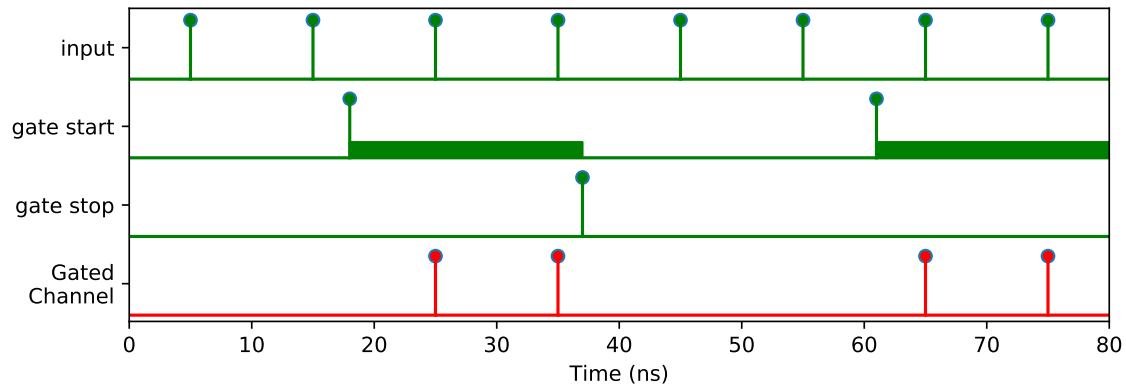
### Protected Member Functions

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

### Additional Inherited Members

#### 9.28.1 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.

## 9.28.2 Constructor & Destructor Documentation

### 9.28.2.1 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 input_channel through the gate
<i>gate_stop_channel</i>	channel on which a signal detected will stop the transmission of the input_channel through the gate
<i>initial</i>	initial state of the gate

### 9.28.2.2 ~GatedChannel()

```
GatedChannel::~GatedChannel ( )
```

### 9.28.3 Member Function Documentation

#### 9.28.3.1 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.

#### 9.28.3.2 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](#).

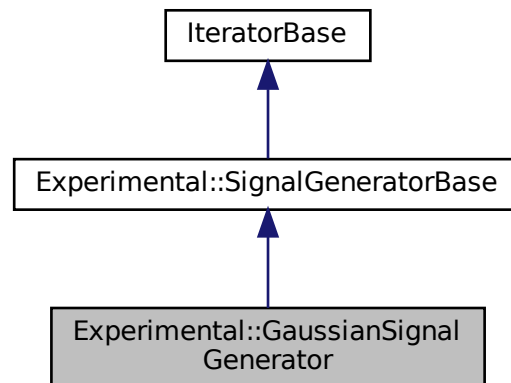
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.29 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](#) ()

## Protected Member Functions

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

## Additional Inherited Members

### 9.29.1 Constructor & Destructor Documentation

#### 9.29.1.1 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.

**9.29.1.2 ~GaussianSignalGenerator()**

Experimental::GaussianSignalGenerator::~~GaussianSignalGenerator ( )

**9.29.2 Member Function Documentation****9.29.2.1 get\_next()**

`timestamp_t` Experimental::GaussianSignalGenerator::get\_next ( ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

**9.29.2.2 initialize()**

void Experimental::GaussianSignalGenerator::initialize (   
 `timestamp_t` *initial\_time* ) [override], [protected], [virtual]

Implements [Experimental::SignalGeneratorBase](#).

**9.29.2.3 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:

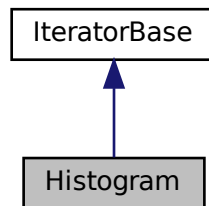
- [Iterators.h](#)

## 9.30 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`< `long long` \*(`size_t`)> `array_out`)

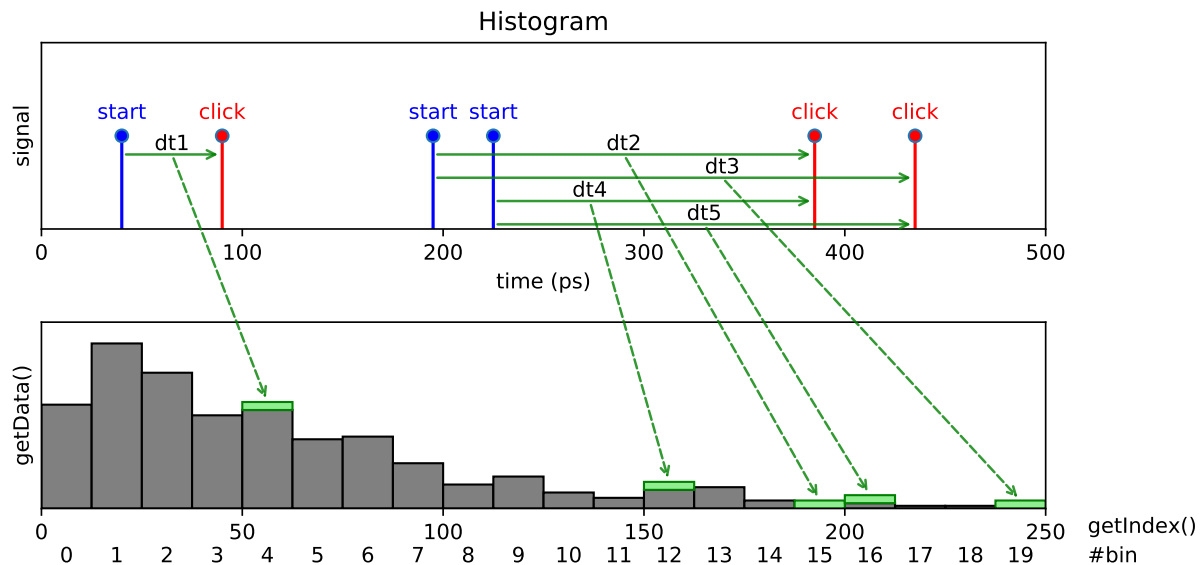
### Protected Member Functions

- 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*

### Additional Inherited Members

#### 9.30.1 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.

## 9.30.2 Constructor & Destructor Documentation

### 9.30.2.1 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

### 9.30.2.2 ~Histogram()

```
Histogram::~~Histogram ( )
```

## 9.30.3 Member Function Documentation

### 9.30.3.1 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](#).

### 9.30.3.2 getData()

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

### 9.30.3.3 getIndex()

```
void Histogram::getIndex (
    std::function< long long *(size_t)> array_out )
```

### 9.30.3.4 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](#).

**9.30.3.5 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:

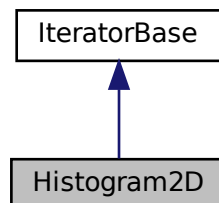
- [Iterators.h](#)

## 9.31 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< long long \*([size\\_t](#), [size\\_t](#), [size\\_t](#))> array\_out)
- void [getIndex\\_1](#) (std::function< long long \*([size\\_t](#))> array\_out)
- void [getIndex\\_2](#) (std::function< long long \*([size\\_t](#))> array\_out)

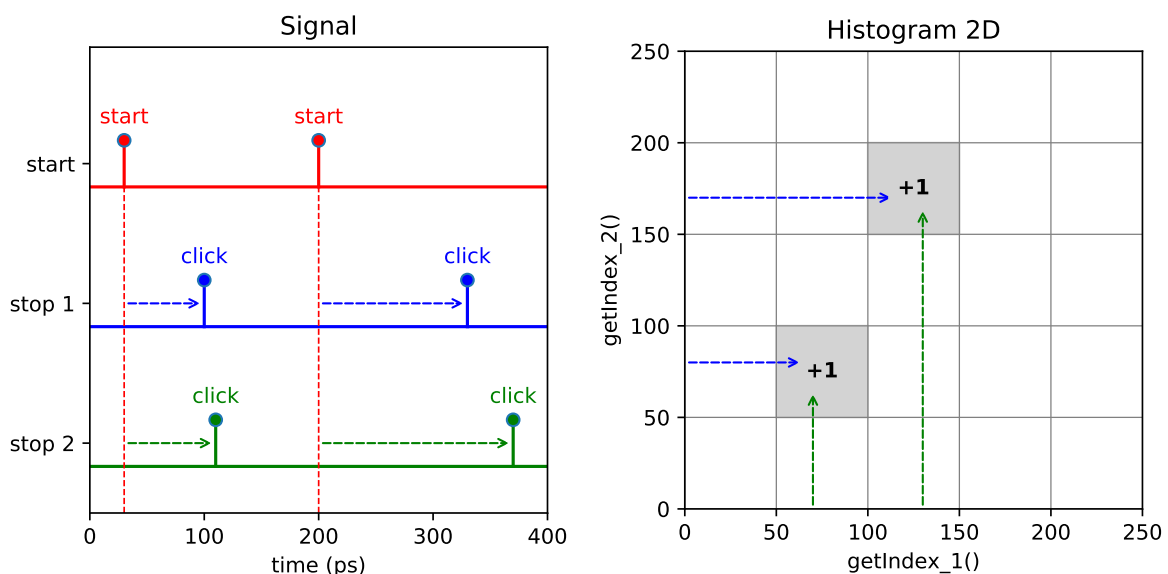
## Protected Member Functions

- 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.*

## Additional Inherited Members

### 9.31.1 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.

## 9.31.2 Constructor & Destructor Documentation

### 9.31.2.1 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

### 9.31.2.2 ~Histogram2D()

```

Histogram2D::~~Histogram2D ( )

```

## 9.31.3 Member Function Documentation

### 9.31.3.1 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](#).

### 9.31.3.2 `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.

### 9.31.3.3 `getIndex()`

```
void Histogram2D::getIndex (
    std::function< long long *(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

### 9.31.3.4 `getIndex_1()`

```
void Histogram2D::getIndex_1 (
    std::function< long long *(size_t)> array_out )
```

Returns a vector of size `n_bins_1` containing the bin locations in ps for the time axis 1.

### 9.31.3.5 `getIndex_2()`

```
void Histogram2D::getIndex_2 (
    std::function< long long *(size_t)> array_out )
```

Returns a vector of size `n_bins_2` containing the bin locations in ps for the time axis 2.

### 9.31.3.6 `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:

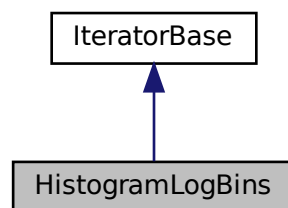
- [Iterators.h](#)

## 9.32 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)  
*constructor of a [HistogramLogBins](#) measurement*
- [~HistogramLogBins](#) ()
- 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< long long \*(size\_t)> array\_out)  
*returns the edges of the bins in ps*

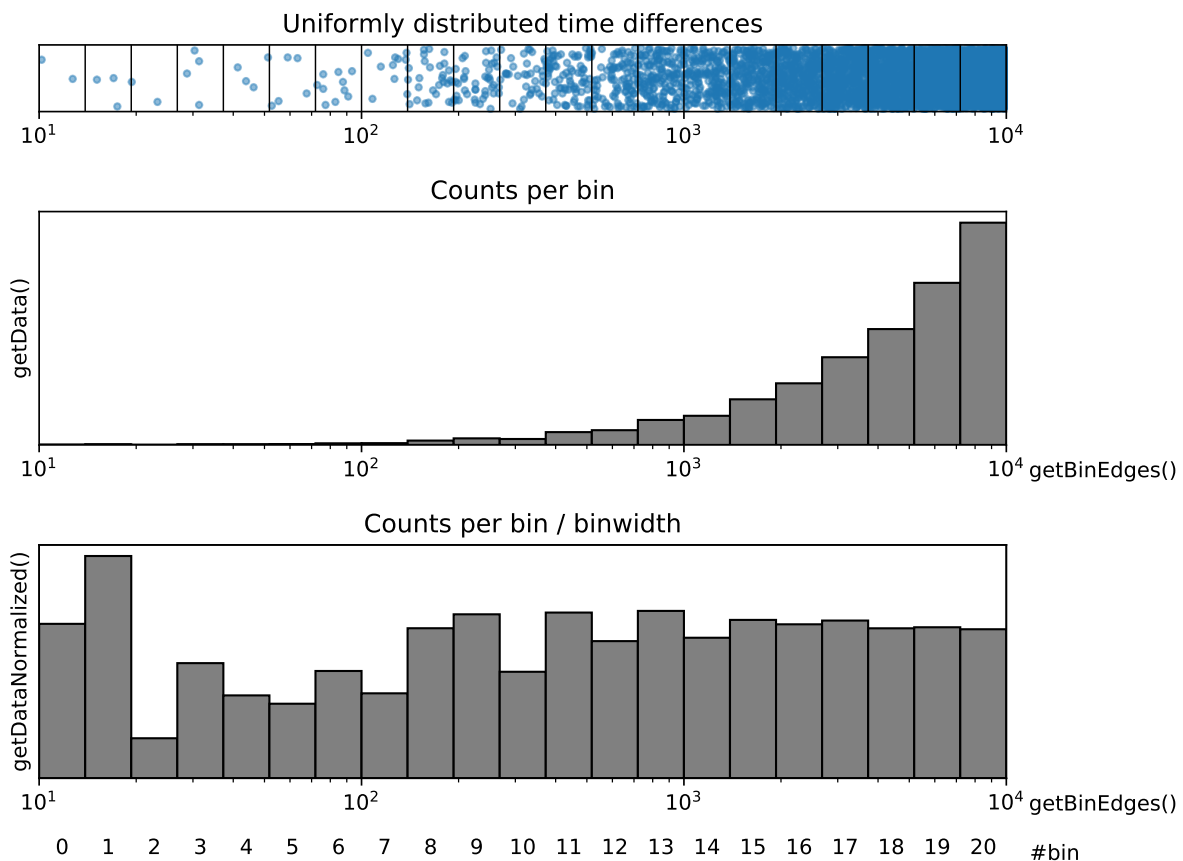
## Protected Member Functions

- 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.*

## Additional Inherited Members

### 9.32.1 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.

### 9.32.2 Constructor & Destructor Documentation

### 9.32.2.1 HistogramLogBins()

```
HistogramLogBins::HistogramLogBins (
    TimeTaggerBase * tagger,
    channel_t click_channel,
    channel_t start_channel,
    double exp_start,
    double exp_stop,
    int32_t n_bins )
```

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
<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

### 9.32.2.2 ~HistogramLogBins()

```
HistogramLogBins::~~HistogramLogBins ( )
```

## 9.32.3 Member Function Documentation

### 9.32.3.1 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](#).

### 9.32.3.2 getBinEdges()

```
void HistogramLogBins::getBinEdges (
    std::function< long long *(size_t)> array_out )
```

returns the edges of the bins in ps

### 9.32.3.3 getData()

```
void HistogramLogBins::getData (
    std::function< uint64_t *(size_t)> array_out )
```

returns the absolute counts for the bins

### 9.32.3.4 getDataNormalizedCountsPerPs()

```
void HistogramLogBins::getDataNormalizedCountsPerPs (
    std::function< double *(size_t)> array_out )
```

returns the counts normalized by the binwidth of each bin

### 9.32.3.5 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](#)

### 9.32.3.6 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:

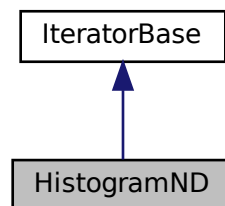
- [Iterators.h](#)

## 9.33 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< long long \*(size\_t)> array\_out, int32\_t dim=0)

## 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.*

## Additional Inherited Members

### 9.33.1 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.

### 9.33.2 Constructor & Destructor Documentation

#### 9.33.2.1 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

#### 9.33.2.2 ~HistogramND()

```
HistogramND::~~HistogramND ( )
```

### 9.33.3 Member Function Documentation

#### 9.33.3.1 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](#).

#### 9.33.3.2 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.

#### 9.33.3.3 getIndex()

```
void HistogramND::getIndex (
    std::function< long long *(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.

#### 9.33.3.4 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:

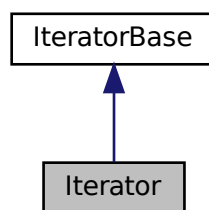
- [Iterators.h](#)

## 9.34 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*

### 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.*

## Additional Inherited Members

### 9.34.1 Detailed Description

a deprecated simple event queue

A simple [Iterator](#), just keeping a first-in first-out queue of event timestamps.

### 9.34.2 Constructor & Destructor Documentation

#### 9.34.2.1 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

#### 9.34.2.2 ~Iterator()

```
Iterator::~~Iterator ( )
```

### 9.34.3 Member Function Documentation

#### 9.34.3.1 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](#).

### 9.34.3.2 next()

```
timestamp_t Iterator::next ( )
```

get next timestamp

get the next timestamp from the queue.

### 9.34.3.3 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](#).

### 9.34.3.4 size()

```
uint64_t Iterator::size ( )
```

get queue size

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

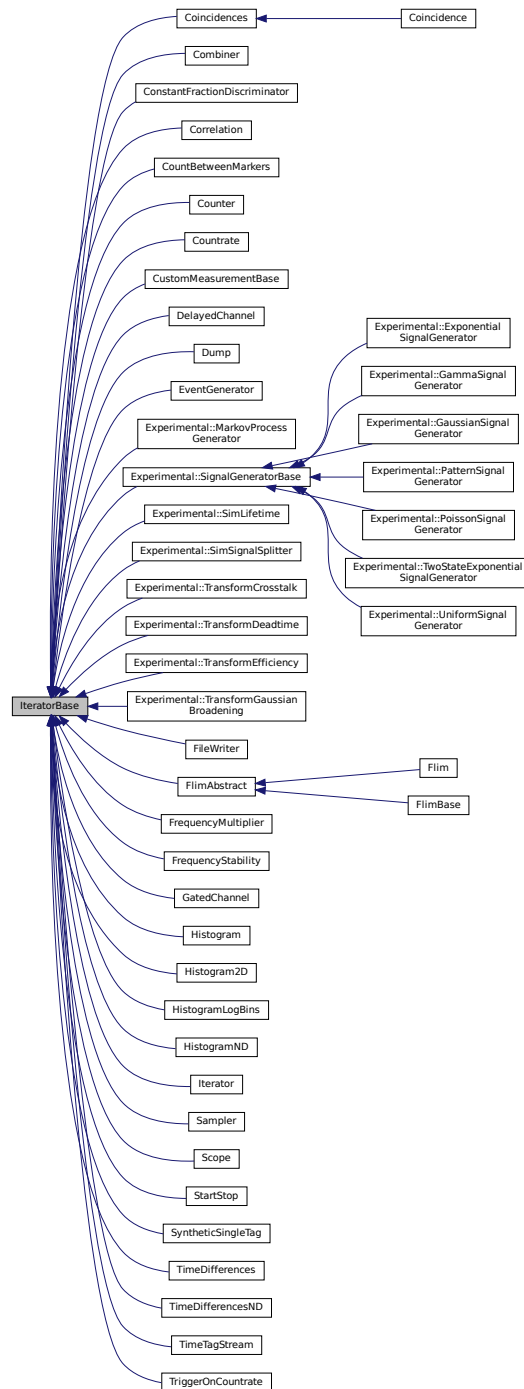
- [Iterators.h](#)

## 9.35 IteratorBase Class Reference

Base class for all iterators.

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

Inheritance diagram for IteratorBase:



## 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.*
- 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.*



## 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.*

### 9.35.1 Detailed Description

Base class for all iterators.

### 9.35.2 Constructor & Destructor Documentation

#### 9.35.2.1 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.

#### 9.35.2.2 ~IteratorBase()

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

destructor, will unregister from the Time Tagger prior finalization.

### 9.35.3 Member Function Documentation

### 9.35.3.1 clear()

```
void IteratorBase::clear ( )
```

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

### 9.35.3.2 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 [FrequencyStability](#), [Sampler](#), [Flim](#), [FlimAbstract](#), [CustomMeasurementBase](#), [EventGenerator](#), [FileWriter](#), [Scope](#), [Correlation](#), [HistogramLogBins](#), [Histogram](#), [TimeDifferencesND](#), [HistogramND](#), [Histogram2D](#), [TimeDifferences](#), [StartStop](#), [Dump](#), [TimeTagStream](#), [Iterator](#), [Countrate](#), [Counter](#), [CountBetweenMarkers](#), and [Combiner](#).

### 9.35.3.3 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` itself if needed.

### 9.35.3.4 finishInitialization()

```
void IteratorBase::finishInitialization ( ) [protected]
```

method to call after finishing the initialization of the measurement

### 9.35.3.5 getCaptureDuration()

```
timestamp_t IteratorBase::getCaptureDuration ( )
```

Total capture duration since the measurement creation or last call to [clear\(\)](#).

Returns

Capture duration in ps

### 9.35.3.6 getConfiguration()

```
std::string IteratorBase::getConfiguration ( )
```

Fetches the overall configuration status of the measurement.

#### Returns

a JSON serialized string with all configuration and status flags.

### 9.35.3.7 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

### 9.35.3.8 getNewVirtualChannel()

```
channel_t IteratorBase::getNewVirtualChannel ( ) [protected]
```

allocate a new virtual output channel for this iterator

### 9.35.3.9 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

### 9.35.3.10 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.

### 9.35.3.11 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 [Experimental::SimLifetime](#), [Experimental::TransformCrosstalk](#), [Experimental::TransformDeadtime](#), [Experimental::TransformGaussianBroadening](#), [Experimental::TransformEfficiency](#), [Experimental::SimSignalSplitter](#), [Experimental::MarkovProcessGenerator](#), [Experimental::SignalGeneratorBase](#), [FrequencyStability](#), [SyntheticSingleTag](#), [Sampler](#), [FlinAbstract](#), [CustomMeasurementBase](#), [EventGenerator](#), [FileWriter](#), [ConstantFractionDiscriminator](#), [Scope](#), [Correlation](#), [HistogramLogBins](#), [Histogram](#), [TimeDifferencesND](#), [HistogramND](#), [Histogram2D](#), [TimeDifferences](#), [StartStop](#), [Dump](#), [TimeTagStream](#), [Iterator](#), [FrequencyMultiplier](#), [GatedChannel](#), [TriggerOnCountrate](#), [DelayedChannel](#), [Countrate](#), [Coincidences](#), [Counter](#), [CountBetweenMarkers](#), and [Combiner](#).

### 9.35.3.12 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 [FrequencyStability](#), [Sampler](#), [FlinAbstract](#), [CustomMeasurementBase](#), [EventGenerator](#), [FileWriter](#), [ConstantFractionDiscriminator](#), [Histogram](#), [TimeDifferencesND](#), [TimeDifferences](#), [StartStop](#), [Dump](#), [TriggerOnCountrate](#), [DelayedChannel](#), [Countrate](#), and [Counter](#).

**9.35.3.13 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 [Experimental::MarkovProcessGenerator](#), [Experimental::SignalGeneratorBase](#), [CustomMeasurementBase](#), [FileWriter](#), and [Dump](#).

**9.35.3.14 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.

**9.35.3.15 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
----------------	-------------

**9.35.3.16 start()**

```
void IteratorBase::start ( )
```

Starts or continues data acquisition.

This method is implicitly called when a measurement object is created.

**9.35.3.17 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

**9.35.3.18 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.

**9.35.3.19 unlock()**

```
void IteratorBase::unlock ( ) [protected]
```

release update lock

see [lock\(\)](#)

**9.35.3.20 unregisterChannel()**

```
void IteratorBase::unregisterChannel (
    channel_t channel ) [protected]
```

unregister a channel

**Parameters**

<i>channel</i>	the channel
----------------	-------------

### 9.35.3.21 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

## 9.35.4 Member Data Documentation

### 9.35.4.1 autostart

```
bool IteratorBase::autostart [protected]
```

Condition if this measurement shall be started by the finishInitialization callback.

### 9.35.4.2 capture\_duration

```
timestamp_t IteratorBase::capture_duration [protected]
```

Duration the iterator has already processed data.

### 9.35.4.3 channels\_registered

```
std::set<channel_t> IteratorBase::channels_registered [protected]
```

list of channels used by the iterator

#### 9.35.4.4 pre\_capture\_duration

```
timestamp_t IteratorBase::pre_capture_duration [protected]
```

For internal use.

#### 9.35.4.5 running

```
bool IteratorBase::running [protected]
```

running state of the iterator

#### 9.35.4.6 tagger

```
TimeTaggerBase* IteratorBase::tagger [protected]
```

Pointer to the corresponding Time Tagger object.

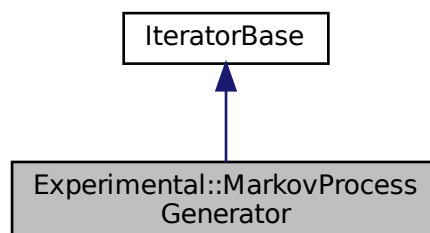
The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.36 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](#) ()

## 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*

## Additional Inherited Members

### 9.36.1 Constructor & Destructor Documentation

#### 9.36.1.1 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 <a href="#">TimeTagger</a>
<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
<i>seed</i>	Seed number for the Pseudo-random number generator. Use -1 to use the current time as seed.

### 9.36.1.2 ~MarkovProcessGenerator()

Experimental::MarkovProcessGenerator::~~MarkovProcessGenerator ( )

## 9.36.2 Member Function Documentation

### 9.36.2.1 getChannel()

channel\_t Experimental::MarkovProcessGenerator::getChannel ( )

### 9.36.2.2 getChannels()

std::vector<channel\_t> Experimental::MarkovProcessGenerator::getChannels ( )

### 9.36.2.3 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](#).

#### 9.36.2.4 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](#)

## 9.37 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](#) ()

#### 9.37.1 Detailed Description

Helper for implementing parallel measurements.

#### 9.37.2 Constructor & Destructor Documentation

##### 9.37.2.1 OrderedBarrier()

```
OrderedBarrier::OrderedBarrier ( )
```

### 9.37.2.2 ~OrderedBarrier()

```
OrderedBarrier::~~OrderedBarrier ( )
```

## 9.37.3 Member Function Documentation

### 9.37.3.1 queue()

```
OrderInstance OrderedBarrier::queue ( )
```

### 9.37.3.2 waitUntilFinished()

```
void OrderedBarrier::waitUntilFinished ( )
```

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

- [TimeTagger.h](#)

## 9.38 OrderedPipeline Class Reference

Helper for implementing parallel measurements.

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

### Public Member Functions

- [OrderedPipeline](#) ()
- [~OrderedPipeline](#) ()

### 9.38.1 Detailed Description

Helper for implementing parallel measurements.

### 9.38.2 Constructor & Destructor Documentation

### 9.38.2.1 OrderedPipeline()

```
OrderedPipeline::OrderedPipeline ( )
```

### 9.38.2.2 ~OrderedPipeline()

```
OrderedPipeline::~~OrderedPipeline ( )
```

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

- [TimeTagger.h](#)

## 9.39 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](#) ()

### 9.39.1 Detailed Description

Internal object for serialization.

### 9.39.2 Constructor & Destructor Documentation

#### 9.39.2.1 OrderInstance() [1/2]

```
OrderedBarrier::OrderInstance::OrderInstance ( )
```

### 9.39.2.2 OrderInstance() [2/2]

```
OrderedBarrier::OrderInstance::OrderInstance (
    OrderedBarrier * parent,
    uint64_t instance_id )
```

### 9.39.2.3 ~OrderInstance()

```
OrderedBarrier::OrderInstance::~~OrderInstance ( )
```

## 9.39.3 Member Function Documentation

### 9.39.3.1 release()

```
void OrderedBarrier::OrderInstance::release ( )
```

### 9.39.3.2 sync()

```
void OrderedBarrier::OrderInstance::sync ( )
```

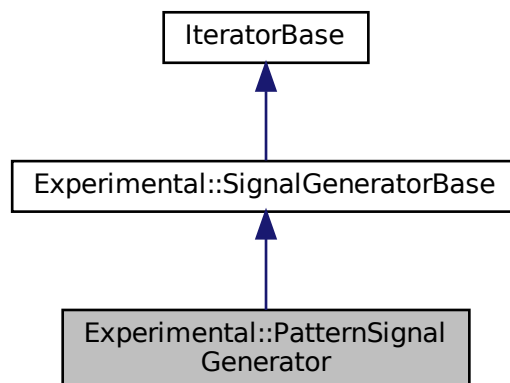
The documentation for this class was generated from the following file:

- [TimeTagger.h](#)

## 9.40 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](#) ()

## Protected Member Functions

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

## Additional Inherited Members

### 9.40.1 Constructor & Destructor Documentation

#### 9.40.1.1 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>
<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.

#### 9.40.1.2 ~PatternSignalGenerator()

```
Experimental::PatternSignalGenerator::~~PatternSignalGenerator ( )
```

## 9.40.2 Member Function Documentation

### 9.40.2.1 `get_next()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.40.2.2 `initialize()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.40.2.3 `on_restart()`

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

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

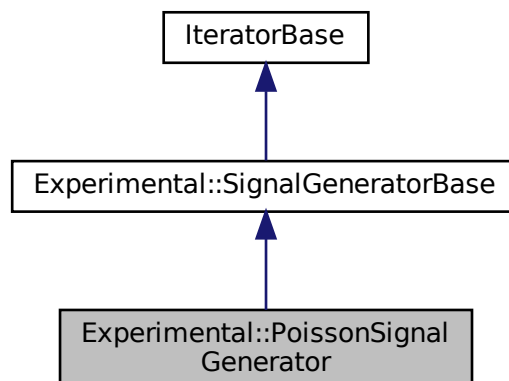
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.41 `Experimental::PoissonSignalGenerator` Class Reference

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

Inheritance diagram for `Experimental::PoissonSignalGenerator`:





## Public Member Functions

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

## Protected Member Functions

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

## Additional Inherited Members

### 9.41.1 Constructor & Destructor Documentation

#### 9.41.1.1 PoissonSignalGenerator()

```
Experimental::PoissonSignalGenerator::PoissonSignalGenerator (
    TimeTaggerBase * tagger,
    double rate,
    channel\_t base_channel = CHANNEL\_UNUSED,
    int32\_t seed = -1 )
```

Construct a poisson event channel.

#### Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>rate</i>	event rate.
<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.

#### 9.41.1.2 ~PoissonSignalGenerator()

```
Experimental::PoissonSignalGenerator::~~PoissonSignalGenerator ( )
```

### 9.41.2 Member Function Documentation

### 9.41.2.1 `get_next()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.41.2.2 `initialize()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.41.2.3 `on_restart()`

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

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

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

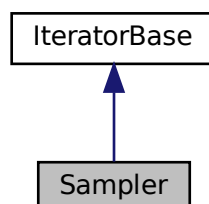
- [Iterators.h](#)

## 9.42 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`< `long long` \*(`size_t`, `size_t`)> array\_out)  
*fetches the internal data as 2D array.*
- `void` `getDataAsMask` (`std::function`< `long long` \*(`size_t`, `size_t`)> array\_out)  
*fetches the internal data as 2D array with a channel mask.*

## 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*

## Additional Inherited Members

### 9.42.1 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.

### 9.42.2 Constructor & Destructor Documentation

#### 9.42.2.1 Sampler()

```
Sampler::Sampler (
    TimeTaggerBase * tagger,
    channel_t trigger,
    std::vector< channel_t > channels,
    size_t max_triggers )
```

constructor of a `Sampler` measurement

## Parameters

<i>tagger</i>	reference to a <a href="#">TimeTagger</a>
<i>trigger</i>	the channel which shall trigger the measurement
<i>channels</i>	a list of channels which will be recorded for every trigger
<i>max_triggers</i>	the maximum amount of triggers without <code>getData*</code> call till this measurement will stop itself

**9.42.2.2 ~Sampler()**

```
Sampler::~Sampler ( )
```

**9.42.3 Member Function Documentation****9.42.3.1 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](#).

**9.42.3.2 getData()**

```
void Sampler::getData (
    std::function< long long *(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)

**9.42.3.3 getDataAsMask()**

```
void Sampler::getDataAsMask (
    std::function< long long *(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

#### 9.42.3.4 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](#).

#### 9.42.3.5 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:

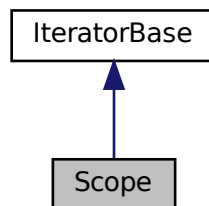
- [Iterators.h](#)

## 9.43 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 ()`

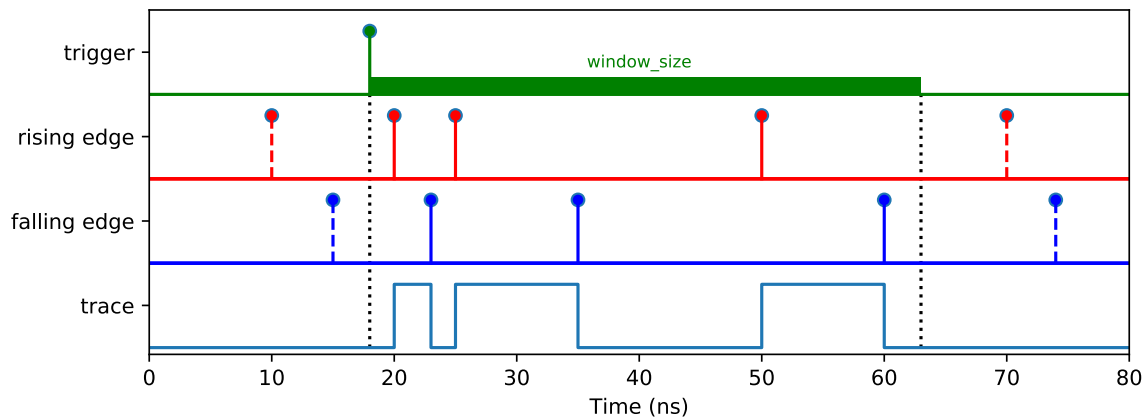
### Protected Member Functions

- `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.*

### Additional Inherited Members

#### 9.43.1 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 `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.

## 9.43.2 Constructor & Destructor Documentation

### 9.43.2.1 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

### 9.43.2.2 ~Scope()

```
Scope::~~Scope ( )
```

## 9.43.3 Member Function Documentation

### 9.43.3.1 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](#).

### 9.43.3.2 getData()

```
std::vector<std::vector<Event> > Scope::getData ( )
```

### 9.43.3.3 getWindowSize()

```
timestamp_t Scope::getWindowSize ( )
```

### 9.43.3.4 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](#).

## 9.43.3.5 ready()

```
bool Scope::ready ( )
```

## 9.43.3.6 triggered()

```
int32_t Scope::triggered ( )
```

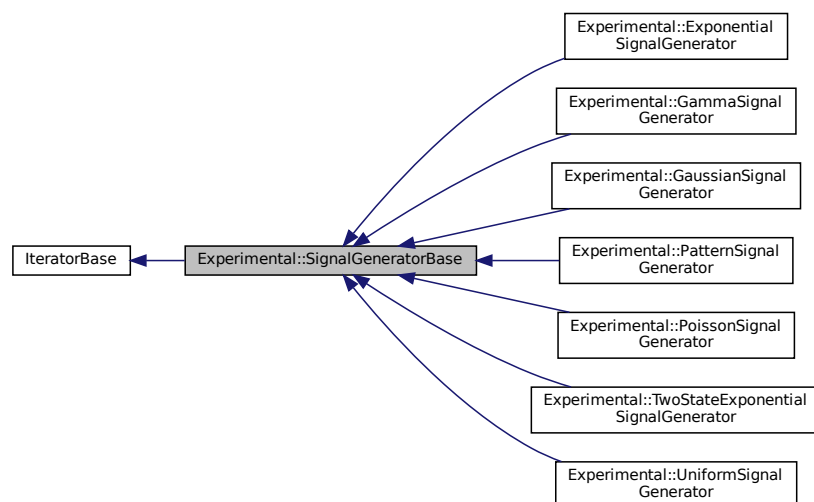
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.44 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*

## 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 Attributes

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

## 9.44.1 Constructor & Destructor Documentation

### 9.44.1.1 SignalGeneratorBase()

```
Experimental::SignalGeneratorBase::SignalGeneratorBase (
    TimeTaggerBase * tagger,
    channel\_t base_channel = CHANNEL\_UNUSED )
```

### 9.44.1.2 ~SignalGeneratorBase()

```
Experimental::SignalGeneratorBase::~SignalGeneratorBase ( )
```

## 9.44.2 Member Function Documentation

#### 9.44.2.1 get\_next()

```
virtual timestamp_t Experimental::SignalGeneratorBase::get_next ( ) [protected], [pure virtual]
```

Implemented in [Experimental::PatternSignalGenerator](#), [Experimental::PoissonSignalGenerator](#), [Experimental::GammaSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::GaussianSignalGenerator](#) and [Experimental::UniformSignalGenerator](#).

#### 9.44.2.2 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.

#### 9.44.2.3 initialize()

```
virtual void Experimental::SignalGeneratorBase::initialize (
    timestamp_t initial_time ) [protected], [pure virtual]
```

Implemented in [Experimental::PatternSignalGenerator](#), [Experimental::PoissonSignalGenerator](#), [Experimental::GammaSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::GaussianSignalGenerator](#) and [Experimental::UniformSignalGenerator](#).

#### 9.44.2.4 isProcessingFinished()

```
bool Experimental::SignalGeneratorBase::isProcessingFinished ( ) [protected]
```

#### 9.44.2.5 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](#).

**9.44.2.6 on\_restart()**

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

Reimplemented in [Experimental::PatternSignalGenerator](#), [Experimental::PoissonSignalGenerator](#), [Experimental::GammaSignalGenerator](#), [Experimental::ExponentialSignalGenerator](#), [Experimental::TwoStateExponentialSignalGenerator](#), [Experimental::GaussianSignalGenerator](#) and [Experimental::UniformSignalGenerator](#).

**9.44.2.7 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](#).

**9.44.2.8 set\_processing\_finished()**

```
void Experimental::SignalGeneratorBase::set_processing_finished (
    bool is_finished ) [protected]
```

**9.44.3 Member Data Documentation**

### 9.44.3.1 impl

```
std::unique_ptr<SignalGeneratorBaseImpl> Experimental::SignalGeneratorBase::impl [protected]
```

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

- [Iterators.h](#)

## 9.45 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 ()

### 9.45.1 Constructor & Destructor Documentation

#### 9.45.1.1 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.

### 9.45.1.2 ~SimDetector()

```
Experimental::SimDetector::~~SimDetector ( )
```

## 9.45.2 Member Function Documentation

### 9.45.2.1 getChannel()

```
channel_t Experimental::SimDetector::getChannel ( )
```

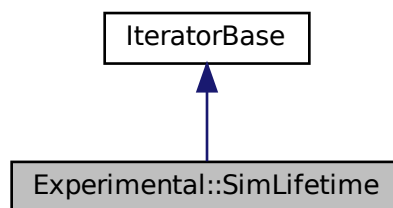
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.46 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)

## Protected Member Functions

- bool [next\\_impl](#) (std::vector< [Tag](#) > &incoming\_tags, [timestamp\\_t](#) begin\_time, [timestamp\\_t](#) end\_time) override

*update iterator state*

## Additional Inherited Members

### 9.46.1 Constructor & Destructor Documentation

#### 9.46.1.1 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.

#### 9.46.1.2 ~SimLifetime()

```
Experimental::SimLifetime::~~SimLifetime ( )
```

### 9.46.2 Member Function Documentation

#### 9.46.2.1 getChannel()

```
channel_t Experimental::SimLifetime::getChannel ( )
```

### 9.46.2.2 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](#).

### 9.46.2.3 registerEmissionReactor()

```
void Experimental::SimLifetime::registerEmissionReactor (
    channel_t trigger_channel,
    std::vector< double > emissions,
    bool repeat )
```

### 9.46.2.4 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:

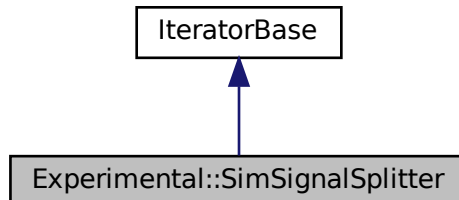
- [Iterators.h](#)



## 9.47 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](#) ()

### Protected Member Functions

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

### Additional Inherited Members

#### 9.47.1 Constructor & Destructor Documentation

##### 9.47.1.1 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.

**9.47.1.2 ~SimSignalSplitter()**

```
Experimental::SimSignalSplitter::~~SimSignalSplitter ( )
```

**9.47.2 Member Function Documentation****9.47.2.1 getChannels()**

```
std::vector<channel_t> Experimental::SimSignalSplitter::getChannels ( )
```

**9.47.2.2 getLeftChannel()**

```
channel_t Experimental::SimSignalSplitter::getLeftChannel ( )
```

**9.47.2.3 getRightChannel()**

```
channel_t Experimental::SimSignalSplitter::getRightChannel ( )
```

**9.47.2.4 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](#)

## 9.48 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](#)

### 9.48.1 Member Data Documentation

#### 9.48.1.1 averaging\_periods

```
double SoftwareClockState::averaging_periods
```

#### 9.48.1.2 clock\_period

`timestamp_t SoftwareClockState::clock_period`

#### 9.48.1.3 enabled

`bool SoftwareClockState::enabled`

#### 9.48.1.4 error\_counter

`uint32_t SoftwareClockState::error_counter`

#### 9.48.1.5 ideal\_clock\_channel

`channel_t SoftwareClockState::ideal_clock_channel`

#### 9.48.1.6 input\_channel

`channel_t SoftwareClockState::input_channel`

#### 9.48.1.7 is\_locked

`bool SoftwareClockState::is_locked`

#### 9.48.1.8 last\_ideal\_clock\_event

`timestamp_t SoftwareClockState::last_ideal_clock_event`

#### 9.48.1.9 period\_error

`double SoftwareClockState::period_error`

## 9.48.1.10 phase\_error\_estimation

```
double SoftwareClockState::phase_error_estimation
```

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

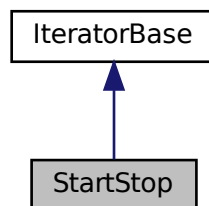
- [TimeTagger.h](#)

## 9.49 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< long long \*(size\_t, size\_t)> [array\\_out](#))

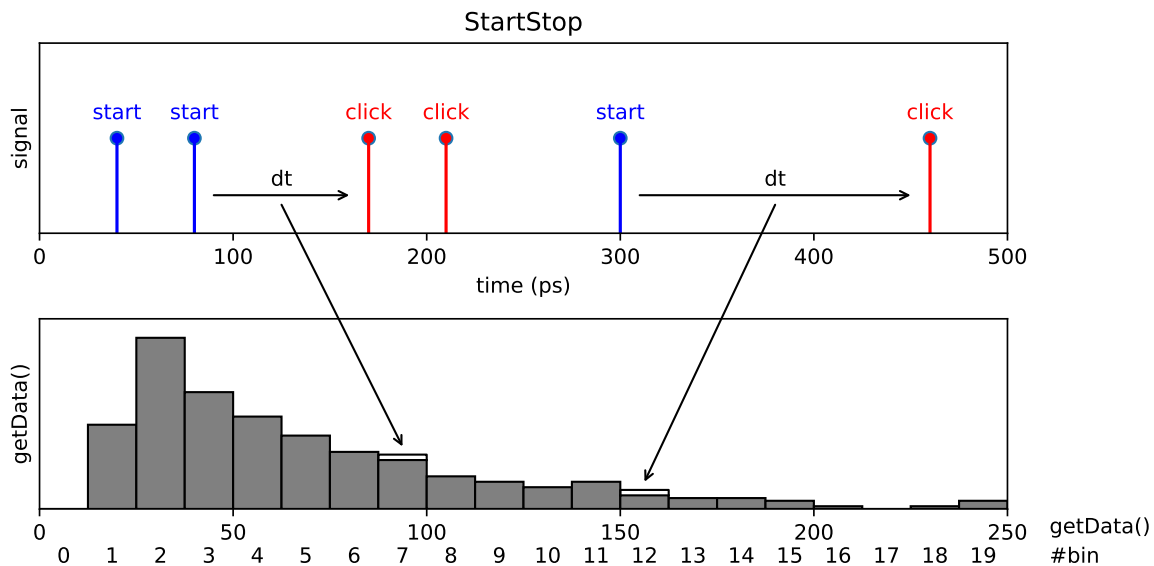
### Protected Member Functions

- 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*

## Additional Inherited Members

### 9.49.1 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.

### 9.49.2 Constructor & Destructor Documentation

#### 9.49.2.1 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

### 9.49.2.2 ~StartStop()

```
StartStop::~~StartStop ( )
```

## 9.49.3 Member Function Documentation

### 9.49.3.1 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](#).

### 9.49.3.2 getData()

```
void StartStop::getData (
    std::function< long long *(size_t, size_t)> array_out )
```

### 9.49.3.3 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](#).

**9.49.3.4 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](#)

**9.50 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*

### 9.50.1 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.

### 9.50.2 Constructor & Destructor Documentation

#### 9.50.2.1 SynchronizedMeasurements()

```
SynchronizedMeasurements::SynchronizedMeasurements (
    TimeTaggerBase * tagger )
```

construct a `SynchronizedMeasurements` object

Parameters

<code>tagger</code>	reference to a <code>TimeTagger</code>
---------------------	--

#### 9.50.2.2 ~SynchronizedMeasurements()

```
SynchronizedMeasurements::~~SynchronizedMeasurements ( )
```

### 9.50.3 Member Function Documentation

#### 9.50.3.1 clear()

```
void SynchronizedMeasurements::clear ( )
```

clear all registered measurements synchronously

### 9.50.3.2 getTagger()

```
TimeTaggerBase* SynchronizedMeasurements::getTagger ( )
```

Returns a proxy tagger object, which shall be used to create immediately registered measurements.

Those measurements will not start automatically.

### 9.50.3.3 isRunning()

```
bool SynchronizedMeasurements::isRunning ( )
```

check if any iterator is running

### 9.50.3.4 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.

### 9.50.3.5 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.

### 9.50.3.6 start()

```
void SynchronizedMeasurements::start ( )
```

start all registered measurements synchronously

**9.50.3.7 startFor()**

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

start all registered measurements synchronously, and stops them after the capture\_duration

**9.50.3.8 stop()**

```
void SynchronizedMeasurements::stop ( )
```

stop all registered measurements synchronously

**9.50.3.9 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.

**9.50.3.10 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:

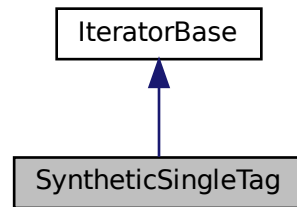
- [Iterators.h](#)

**9.51 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

## Protected Member Functions

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

## Additional Inherited Members

### 9.51.1 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.

### 9.51.2 Constructor & Destructor Documentation

#### 9.51.2.1 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.

## 9.51.2.2 ~SyntheticSingleTag()

```
SyntheticSingleTag::~SyntheticSingleTag ( )
```

## 9.51.3 Member Function Documentation

## 9.51.3.1 getChannel()

```
channel_t SyntheticSingleTag::getChannel ( ) const
```

## 9.51.3.2 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](#).

### 9.51.3.3 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](#)

## 9.52 Tag Struct Reference

a single event on a channel

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

### Public Types

- enum [Type](#) : unsigned char {  
[Type::TimeTag](#) = 0, [Type::Error](#) = 1, [Type::OverflowBegin](#) = 2, [Type::OverflowEnd](#) = 3,  
[Type::MissedEvents](#) = 4 }

*This enum marks what kind of event this object represents.*

### Public Attributes

- enum [Tag::Type](#) type
- 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*

### 9.52.1 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.

### 9.52.2 Member Enumeration Documentation

### 9.52.2.1 Type

```
enum 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	

## 9.52.3 Member Data Documentation

### 9.52.3.1 channel

```
channel_t Tag::channel
```

the channel number

### 9.52.3.2 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.

### 9.52.3.3 reserved

```
char Tag::reserved
```

8 bit padding

Reserved for future use. Set it to zero.

### 9.52.3.4 time

```
timestamp_t Tag::time
```

the timestamp of the event in picoseconds

### 9.52.3.5 type

```
enum Tag::Type Tag::type
```

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

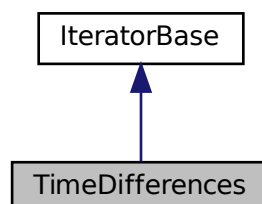
- [TimeTagger.h](#)

## 9.53 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< long long \*([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*

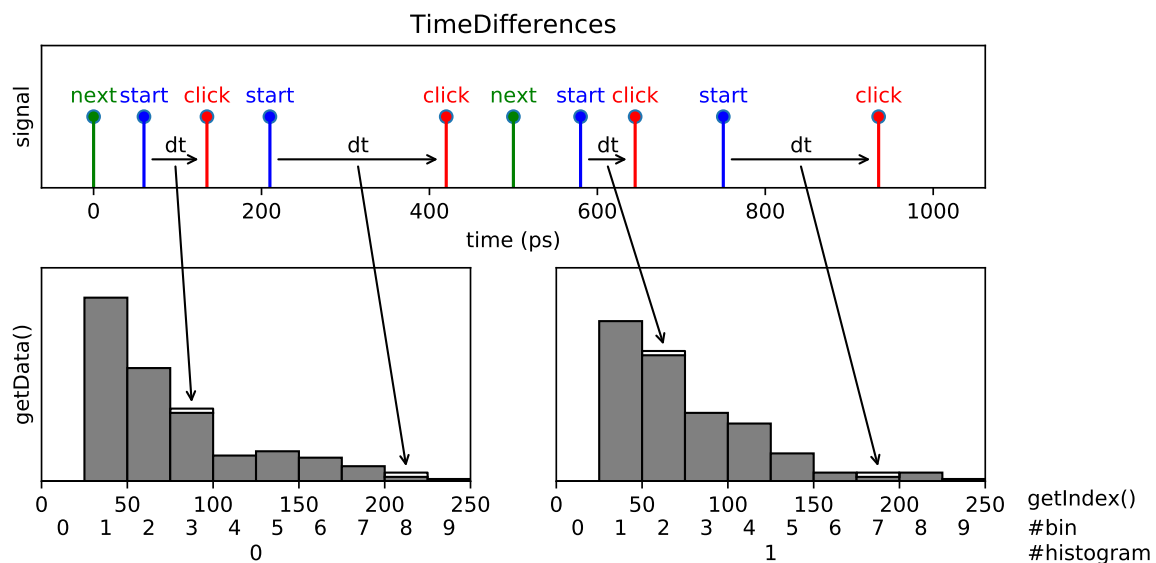
## Protected Member Functions

- 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*

## Additional Inherited Members

### 9.53.1 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.

## 9.53.2 Constructor & Destructor Documentation

### 9.53.2.1 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

### 9.53.2.2 ~TimeDifferences()

```
TimeDifferences::~~TimeDifferences ( )
```

## 9.53.3 Member Function Documentation

### 9.53.3.1 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](#).

### 9.53.3.2 getCounts()

```
uint64_t TimeDifferences::getCounts ( )
```

returns the number of rollovers (histogram index resets)

### 9.53.3.3 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

### 9.53.3.4 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

### 9.53.3.5 getIndex()

```
void TimeDifferences::getIndex (
    std::function< long long *(size_t)> array_out )
```

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

### 9.53.3.6 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](#).

### 9.53.3.7 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](#).

### 9.53.3.8 ready()

```
bool TimeDifferences::ready ( )
```

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

### 9.53.3.9 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](#)

## 9.54 TimeDifferencesImpl< T > Class Template Reference

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

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

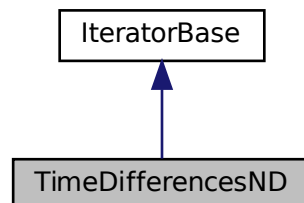
- [Iterators.h](#)

## 9.55 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\_histograms, `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_histograms` containing the histograms*
- `void getIndex` (`std::function< long long *(size_t)>` array\_out)  
*returns a vector of size `n_bins` containing the time bins in ps*

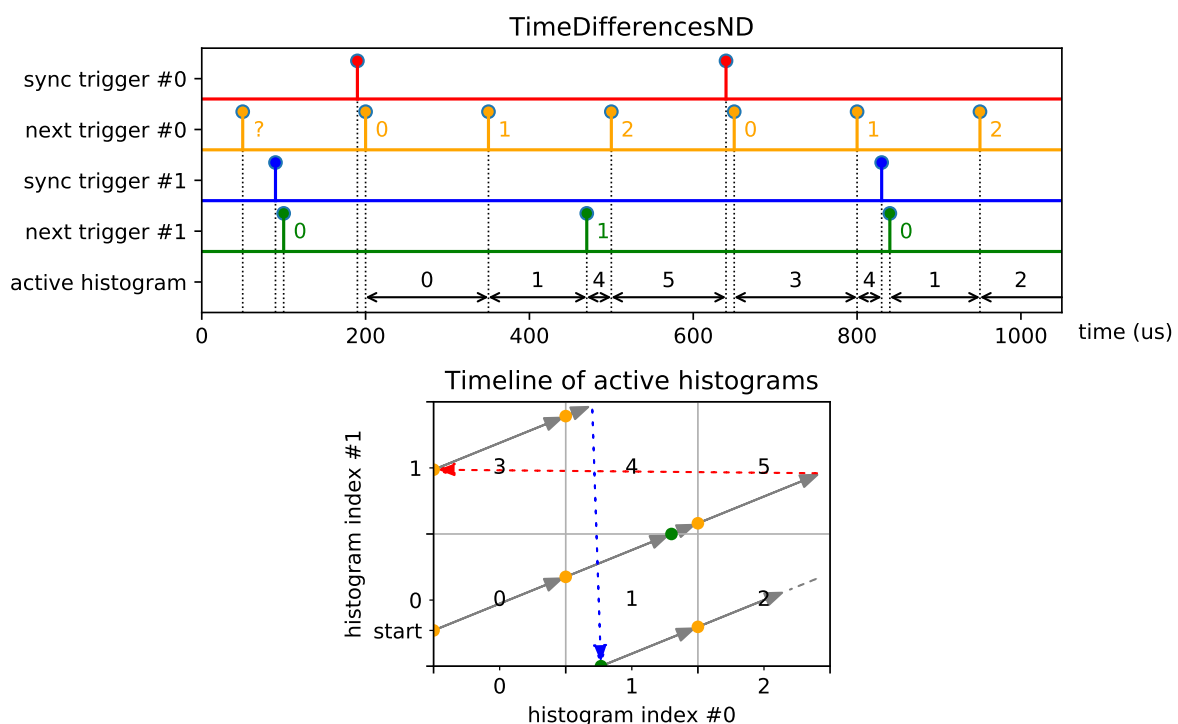
## Protected Member Functions

- `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*

## Additional Inherited Members

### 9.55.1 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

## 9.55.2 Constructor & Destructor Documentation

### 9.55.2.1 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

### 9.55.2.2 ~TimeDifferencesND()

```
TimeDifferencesND::~~TimeDifferencesND ( )
```

## 9.55.3 Member Function Documentation

### 9.55.3.1 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](#).

### 9.55.3.2 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

### 9.55.3.3 getIndex()

```
void TimeDifferencesND::getIndex (
    std::function< long long *(size_t)> array_out )
```

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

### 9.55.3.4 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>	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 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:

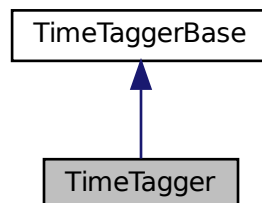
- [Iterators.h](#)

**9.56 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 [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 eletrically 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](#) interface=[FpgaLinkInterface::SFPP\\_10GE](#))=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*

## Additional Inherited Members

### 9.56.1 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.

### 9.56.2 Member Function Documentation

#### 9.56.2.1 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.

### 9.56.2.2 clearConditionalFilter()

```
virtual void TimeTagger::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to `setConditionalFilter({}, {})`

### 9.56.2.3 disableFpgaLink()

```
virtual void TimeTagger::disableFpgaLink ( ) [pure virtual]
```

Disable the FPGA link of the Time Tagger X.

### 9.56.2.4 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
-----------------	------------------------------------

### 9.56.2.5 enableFpgaLink()

```
virtual void TimeTagger::enableFpgaLink (
    std::vector< channel_t > channels,
    std::string destination_mac,
    FpgaLinkInterface interface = FpgaLinkInterface::SFPP_10GE ) [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 broadcast
<i>interface</i>	selects which interface shall be used, default is <code>FpgaLinkInterface::SFPP_10GE</code>

### 9.56.2.6 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.

### 9.56.2.7 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 inverted Channels of the rising edges.

### 9.56.2.8 getChannelNumberScheme()

```
virtual int TimeTagger::getChannelNumberScheme ( ) [pure virtual]
```

Fetch the configured numbering scheme for this [TimeTagger](#) object.

Please see [setTimeTaggerChannelNumberScheme\(\)](#) for details.

### 9.56.2.9 getConditionalFilterFiltered()

```
virtual std::vector<channel_t> TimeTagger::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

### 9.56.2.10 getConditionalFilterTrigger()

```
virtual std::vector<channel_t> TimeTagger::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

#### 9.56.2.11 getDACRange()

```
virtual std::vector<double> TimeTagger::getDACRange ( ) [pure virtual]
```

returns the minimum and the maximum voltage of the DACs as a trigger reference

#### 9.56.2.12 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

#### 9.56.2.13 getDistributionCount()

```
virtual void TimeTagger::getDistributionCount (
    std::function< uint64_t *(size_t, size_t)> array_out ) [pure virtual]
```

get internal calibration data

#### 9.56.2.14 getDistributionPSEcs()

```
virtual void TimeTagger::getDistributionPSEcs (
    std::function< double *(size_t, size_t)> array_out ) [pure virtual]
```

get internal calibration data

#### 9.56.2.15 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

**9.56.2.16 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"

**9.56.2.17 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

**9.56.2.18 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

**9.56.2.19 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

**9.56.2.20 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

**9.56.2.21 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

**9.56.2.22 getModel()**

```
virtual std::string TimeTagger::getModel ( ) [pure virtual]
```

identifies the hardware by Time Tagger Model

**9.56.2.23 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

**9.56.2.24 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.

#### 9.56.2.25 getPsPerClock()

```
virtual timestamp_t TimeTagger::getPsPerClock ( ) [pure virtual]
```

fetch the duration of each clock cycle in picoseconds

#### 9.56.2.26 getSensorData()

```
virtual std::string TimeTagger::getSensorData ( ) [pure virtual]
```

Show the status of the sensor data from the FPGA and peripherals on the console.

#### 9.56.2.27 getSerial()

```
virtual std::string TimeTagger::getSerial ( ) [pure virtual]
```

identifies the hardware by serial number

#### 9.56.2.28 getStreamBlockSizeEvents()

```
virtual int TimeTagger::getStreamBlockSizeEvents ( ) [pure virtual]
```

#### 9.56.2.29 getStreamBlockSizeLatency()

```
virtual int TimeTagger::getStreamBlockSizeLatency ( ) [pure virtual]
```

#### 9.56.2.30 getTestSignalDivider()

```
virtual int TimeTagger::getTestSignalDivider ( ) [pure virtual]
```

get the divider for the frequency of the test signal

#### 9.56.2.31 getTriggerLevel()

```
virtual double TimeTagger::getTriggerLevel (
    channel_t channel ) [pure virtual]
```

get the trigger voltage threshold of a channel

## Parameters

<i>channel</i>	the channel
----------------	-------------

**9.56.2.32 isChannelRegistered()**

```
virtual bool TimeTagger::isChannelRegistered (
    channel_t chan ) [pure virtual]
```

**9.56.2.33 isServerRunning()**

```
virtual bool TimeTagger::isServerRunning ( ) [pure virtual]
```

check if the server is still running.

## Returns

returns true if running; false, if not running

**9.56.2.34 reset()**

```
virtual void TimeTagger::reset ( ) [pure virtual]
```

reset the [TimeTagger](#) object to default settings and detach all iterators

**9.56.2.35 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

**9.56.2.36 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  $n$ 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 smaller than 65536

**9.56.2.37 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
-------------	-----------------------------------

**9.56.2.38 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

#### 9.56.2.39 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

#### 9.56.2.40 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

### 9.56.2.41 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.

### 9.56.2.42 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

### 9.56.2.43 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
----------------	-------------------------------

### 9.56.2.44 setStreamBlockSize()

```
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

#### 9.56.2.45 setTestSignalDivider()

```
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
----------------	-------------------------------------

#### 9.56.2.46 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.
---------------	--

#### 9.56.2.47 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]

**9.56.2.48 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.

**9.56.2.49 stopServer()**

```
virtual void TimeTagger::stopServer ( ) [pure virtual]
```

stops the time tagger server if currently running, otherwise does nothing.

**9.56.2.50 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

**9.56.2.51 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

**9.56.2.52 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

**9.56.2.53 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

**9.56.2.54 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

**9.56.2.55 xtra\_measureTriggerLevel()**

```
virtual double TimeTagger::xtra_measureTriggerLevel (
    channel_t channel ) [pure virtual]
```

measures the eletrically 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

**9.56.2.56 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

**9.56.2.57 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

**9.56.2.58 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
----------------	---

**9.56.2.59 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
----------------	---

**9.56.2.60 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
---------------	--------------------------

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

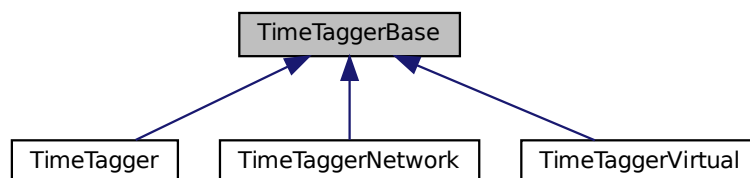
- [TimeTagger.h](#)

**9.57 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 raising 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

### 9.57.1 Detailed Description

Basis interface for all Time Tagger classes.

This basis interface represents all common methods to add, remove, and run measurements.

### 9.57.2 Member Typedef Documentation

#### 9.57.2.1 IteratorCallback

```
typedef std::function<void(IteratorBase *)> TimeTaggerBase::IteratorCallback
```

#### 9.57.2.2 IteratorCallbackMap

```
typedef std::map<IteratorBase *, IteratorCallback> TimeTaggerBase::IteratorCallbackMap
```

### 9.57.3 Constructor & Destructor Documentation

### 9.57.3.1 TimeTaggerBase() [1/2]

```
TimeTaggerBase::TimeTaggerBase ( ) [inline], [protected]
```

abstract interface class

### 9.57.3.2 ~TimeTaggerBase()

```
virtual TimeTaggerBase::~~TimeTaggerBase ( ) [inline], [protected], [virtual]
```

destructor

### 9.57.3.3 TimeTaggerBase() [2/2]

```
TimeTaggerBase::TimeTaggerBase (
    const TimeTaggerBase & ) [protected], [delete]
```

## 9.57.4 Member Function Documentation

### 9.57.4.1 addChild()

```
virtual void TimeTaggerBase::addChild (
    TimeTaggerBase * child ) [protected], [pure virtual]
```

### 9.57.4.2 addIterator()

```
virtual std::shared_ptr<IteratorBaseListNode> TimeTaggerBase::addIterator (
    IteratorBase * it ) [protected], [pure virtual]
```

### 9.57.4.3 clearOverflows()

```
virtual void TimeTaggerBase::clearOverflows ( ) [pure virtual]
```

clear overflow counter

Sets the overflow counter to zero

#### 9.57.4.4 disableSoftwareClock()

```
virtual void TimeTaggerBase::disableSoftwareClock ( ) [pure virtual]
```

disabled the software PLL

See setSoftwareClock for further details.

#### 9.57.4.5 freeIterator()

```
virtual void TimeTaggerBase::freeIterator (
    IteratorBase * it ) [protected], [pure virtual]
```

#### 9.57.4.6 freeVirtualChannel()

```
virtual void TimeTaggerBase::freeVirtualChannel (
    channel_t channel ) [protected], [pure virtual]
```

#### 9.57.4.7 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.

#### 9.57.4.8 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

**9.57.4.9 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

**9.57.4.10 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

**9.57.4.11 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 `waitForFence` call were actually produced after the `getFence` call. The `waitForFence` function waits until all tags, which are present at the time of the function call within the internal memory of the Time Tagger, are processed. This call might block to limit the 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

#### 9.57.4.12 `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

#### 9.57.4.13 `getInvertedChannel()`

```
virtual channel_t TimeTaggerBase::getInvertedChannel (
    channel_t channel ) [pure virtual]
```

get the falling channel id for a raising 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

**9.57.4.14 getNewVirtualChannel()**

```
virtual channel_t TimeTaggerBase::getNewVirtualChannel ( ) [protected], [pure virtual]
```

**9.57.4.15 getOverflows()**

```
virtual long long TimeTaggerBase::getOverflows ( ) [pure virtual]
```

get overflow count

Get the number of communication overflows occurred

**9.57.4.16 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

**9.57.4.17 getSoftwareClockState()**

```
virtual SoftwareClockState TimeTaggerBase::getSoftwareClockState ( ) [pure virtual]
```

queries all state information of the software clock

See setSoftwareClock for further details.

**9.57.4.18 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](#).

#### 9.57.4.19 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

#### 9.57.4.20 operator=()

```
TimeTaggerBase& TimeTaggerBase::operator= (
    const TimeTaggerBase & ) [protected], [delete]
```

#### 9.57.4.21 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
----------------	-------------

#### 9.57.4.22 registerChannel() [2/2]

```
virtual void TimeTaggerBase::registerChannel (
    std::set< channel_t > channels ) [protected], [pure virtual]
```

#### 9.57.4.23 release()

```
virtual void TimeTaggerBase::release ( ) [protected], [pure virtual]
```



#### 9.57.4.24 removeChild()

```
virtual void TimeTaggerBase::removeChild (
    TimeTaggerBase * child ) [protected], [pure virtual]
```

#### 9.57.4.25 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

#### 9.57.4.26 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

**9.57.4.27 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

**9.57.4.28 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

**9.57.4.29 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

#### 9.57.4.30 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.

#### 9.57.4.31 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

### 9.57.4.32 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

### 9.57.4.33 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

**9.57.4.34 unregisterChannel() [1/2]**

```
virtual void TimeTaggerBase::unregisterChannel (
    channel_t channel ) [protected], [pure virtual]
```

release a previously registered channel.

**Parameters**

<i>channel</i>	the channel
----------------	-------------

**9.57.4.35 unregisterChannel() [2/2]**

```
virtual void TimeTaggerBase::unregisterChannel (
    std::set< channel_t > channels ) [protected], [pure virtual]
```

**9.57.4.36 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:

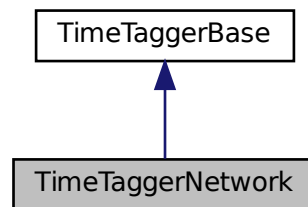
- [TimeTagger.h](#)

**9.58 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*

## Additional Inherited Members

### 9.58.1 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.

### 9.58.2 Member Function Documentation

#### 9.58.2.1 clearConditionalFilter()

```
virtual void TimeTaggerNetwork::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to `setConditionalFilter({},{})`

#### 9.58.2.2 clearOverflowsClient()

```
virtual void TimeTaggerNetwork::clearOverflowsClient ( ) [pure virtual]
```

#### 9.58.2.3 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 inverted Channels of the rising edges.

#### 9.58.2.4 getChannelNumberScheme()

```
virtual int TimeTaggerNetwork::getChannelNumberScheme ( ) [pure virtual]
```

Fetch the configured numbering scheme for this [TimeTagger](#) object.

Please see [setTimeTaggerChannelNumberScheme\(\)](#) for details.



#### 9.58.2.5 getConditionalFilterFiltered()

```
virtual std::vector<channel_t> TimeTaggerNetwork::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

#### 9.58.2.6 getConditionalFilterTrigger()

```
virtual std::vector<channel_t> TimeTaggerNetwork::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see setConditionalFilter

#### 9.58.2.7 getDACRange()

```
virtual std::vector<double> TimeTaggerNetwork::getDACRange ( ) [pure virtual]
```

returns the minimum and the maximum voltage of the DACs as a trigger reference

#### 9.58.2.8 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

#### 9.58.2.9 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

**9.58.2.10 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

**9.58.2.11 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"

**9.58.2.12 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

### 9.58.2.13 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

### 9.58.2.14 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

### 9.58.2.15 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

**9.58.2.16 getModel()**

```
virtual std::string TimeTaggerNetwork::getModel ( ) [pure virtual]
```

identifies the hardware by Time Tagger Model

**9.58.2.17 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

**9.58.2.18 getOverflowsAndClearClient()**

```
virtual long long TimeTaggerNetwork::getOverflowsAndClearClient ( ) [pure virtual]
```

**9.58.2.19 getOverflowsClient()**

```
virtual long long TimeTaggerNetwork::getOverflowsClient ( ) [pure virtual]
```

#### 9.58.2.20 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.

#### 9.58.2.21 getPsPerClock()

```
virtual timestamp_t TimeTaggerNetwork::getPsPerClock ( ) [pure virtual]
```

fetch the duration of each clock cycle in picoseconds

#### 9.58.2.22 getSensorData()

```
virtual std::string TimeTaggerNetwork::getSensorData ( ) [pure virtual]
```

Show the status of the sensor data from the FPGA and peripherals on the console.

#### 9.58.2.23 getSerial()

```
virtual std::string TimeTaggerNetwork::getSerial ( ) [pure virtual]
```

identifies the hardware by serial number

#### 9.58.2.24 getStreamBlockSizeEvents()

```
virtual int TimeTaggerNetwork::getStreamBlockSizeEvents ( ) [pure virtual]
```

#### 9.58.2.25 getStreamBlockSizeLatency()

```
virtual int TimeTaggerNetwork::getStreamBlockSizeLatency ( ) [pure virtual]
```

#### 9.58.2.26 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](#).

**9.58.2.27 getTestSignalDivider()**

```
virtual int TimeTaggerNetwork::getTestSignalDivider ( ) [pure virtual]
```

get the divider for the frequency of the test signal

**9.58.2.28 getTriggerLevel()**

```
virtual double TimeTaggerNetwork::getTriggerLevel (
    channel\_t channel ) [pure virtual]
```

get the trigger voltage threshold of a channel

## Parameters

<i>channel</i>	the channel
----------------	-------------

**9.58.2.29 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.

**9.58.2.30 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

**9.58.2.31 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

**9.58.2.32 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 smaller than 65536



### 9.58.2.33 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
-------------	-----------------------------------

### 9.58.2.34 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

### 9.58.2.35 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

**9.58.2.36 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.

**9.58.2.37 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

**9.58.2.38 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
----------------	-------------------------------

**9.58.2.39 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

**9.58.2.40 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
----------------	-------------------------------------

**9.58.2.41 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 compressio is active or not.
---------------	---

**9.58.2.42 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:

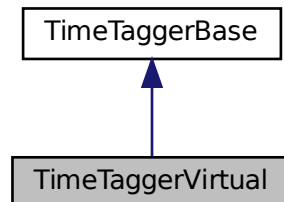
- [TimeTagger.h](#)

## 9.59 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*

## Additional Inherited Members

### 9.59.1 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.

### 9.59.2 Member Function Documentation

#### 9.59.2.1 clearConditionalFilter()

```
virtual void TimeTaggerVirtual::clearConditionalFilter ( ) [pure virtual]
```

deactivates the conditional filter

equivalent to `setConditionalFilter({},{})`

#### 9.59.2.2 getConditionalFilterFiltered()

```
virtual std::vector<channel_t> TimeTaggerVirtual::getConditionalFilterFiltered ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see `setConditionalFilter`

#### 9.59.2.3 getConditionalFilterTrigger()

```
virtual std::vector<channel_t> TimeTaggerVirtual::getConditionalFilterTrigger ( ) [pure virtual]
```

fetches the configuration of the conditional filter

see `setConditionalFilter`

#### 9.59.2.4 getReplaySpeed()

```
virtual double TimeTaggerVirtual::getReplaySpeed ( ) [pure virtual]
```

fetches the speed factor

Please see `setReplaySpeed` for more details.

#### Returns

the speed factor

### 9.59.2.5 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

### 9.59.2.6 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.

### 9.59.2.7 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

### 9.59.2.8 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
--------------	---

### 9.59.2.9 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.

### 9.59.2.10 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:

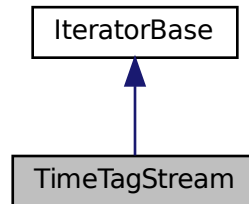
- [TimeTagger.h](#)

## 9.60 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*

### 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.*

### Additional Inherited Members

#### 9.60.1 Detailed Description

access the time tag stream

#### 9.60.2 Constructor & Destructor Documentation



### 9.60.2.1 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

### 9.60.2.2 ~TimeTagStream()

```
TimeTagStream::~~TimeTagStream ( )
```

## 9.60.3 Member Function Documentation

### 9.60.3.1 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](#).

### 9.60.3.2 getCounts()

```
uint64_t TimeTagStream::getCounts ( )
```

return the number of stored tags

### 9.60.3.3 `getData()`

```
TimeTagStreamBuffer TimeTagStream::getData ( )
```

fetches all stored tags and clears the internal state

### 9.60.3.4 `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](#)

## 9.61 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< int \*(size\_t)> array\_out)
- void [getTimestamps](#) (std::function< long long \*(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](#)

### 9.61.1 Detailed Description

return object for [TimeTagStream::getData](#)

### 9.61.2 Constructor & Destructor Documentation

#### 9.61.2.1 ~TimeTagStreamBuffer()

```
TimeTagStreamBuffer::~TimeTagStreamBuffer ( )
```

### 9.61.3 Member Function Documentation

#### 9.61.3.1 getChannels()

```
void TimeTagStreamBuffer::getChannels (
    std::function< int *(size_t)> array_out )
```

#### 9.61.3.2 getEventTypes()

```
void TimeTagStreamBuffer::getEventTypes (
    std::function< unsigned char *(size_t)> array_out )
```

#### 9.61.3.3 getMissedEvents()

```
void TimeTagStreamBuffer::getMissedEvents (
    std::function< unsigned short *(size_t)> array_out )
```

#### 9.61.3.4 getOverflows()

```
void TimeTagStreamBuffer::getOverflows (
    std::function< unsigned char *(size_t)> array_out )
```

#### 9.61.3.5 getTimestamps()

```
void TimeTagStreamBuffer::getTimestamps (
    std::function< long long *(size_t)> array_out )
```

### 9.61.4 Member Data Documentation

#### 9.61.4.1 hasOverflows

```
bool TimeTagStreamBuffer::hasOverflows
```

#### 9.61.4.2 size

```
uint64_t TimeTagStreamBuffer::size
```

#### 9.61.4.3 tGetData

```
timestamp_t TimeTagStreamBuffer::tGetData
```

#### 9.61.4.4 tStart

```
timestamp_t TimeTagStreamBuffer::tStart
```

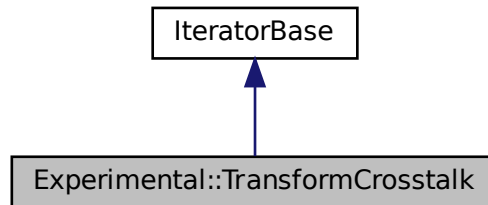
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.62 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 ()

### Protected Member Functions

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

### Additional Inherited Members

#### 9.62.1 Constructor & Destructor Documentation

##### 9.62.1.1 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.

9.62.1.2 `~TransformCrosstalk()`

```
Experimental::TransformCrosstalk::~~TransformCrosstalk ( )
```

## 9.62.2 Member Function Documentation

9.62.2.1 `getChannel()`

```
channel_t Experimental::TransformCrosstalk::getChannel ( )
```

9.62.2.2 `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](#).

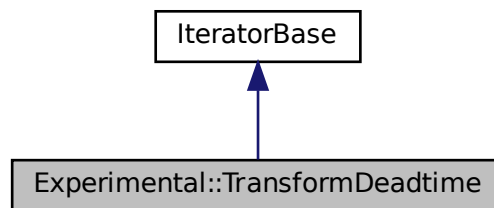
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.63 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 ()

**Protected Member Functions**

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

**Additional Inherited Members**

### 9.63.1 Constructor & Destructor Documentation

### 9.63.1.1 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.

### 9.63.1.2 ~TransformDeadtime()

```
Experimental::TransformDeadtime::~~TransformDeadtime ( )
```

## 9.63.2 Member Function Documentation

### 9.63.2.1 getChannel()

```
channel_t Experimental::TransformDeadtime::getChannel ( )
```

### 9.63.2.2 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](#).

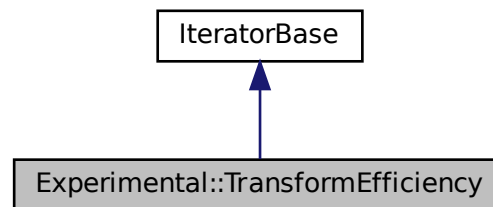
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.64 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 effcincy of 1.0 won't drop any events, an efficiency of 0.5 will drop half the events.
- [~TransformEfficiency](#) ()
- [channel\\_t](#) getChannel ()

### Protected Member Functions

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

## Additional Inherited Members

### 9.64.1 Constructor & Destructor Documentation

#### 9.64.1.1 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.

#### 9.64.1.2 ~TransformEfficiency()

```
Experimental::TransformEfficiency::~~TransformEfficiency ( )
```

### 9.64.2 Member Function Documentation

#### 9.64.2.1 getChannel()

```
channel_t Experimental::TransformEfficiency::getChannel ( )
```

### 9.64.2.2 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](#).

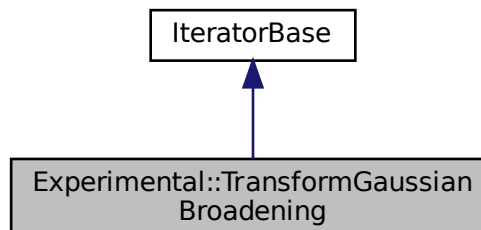
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.65 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 ()

## Protected Member Functions

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

## Additional Inherited Members

### 9.65.1 Constructor & Destructor Documentation

#### 9.65.1.1 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.

### 9.65.1.2 ~TransformGaussianBroadening()

```
Experimental::TransformGaussianBroadening::~~TransformGaussianBroadening ( )
```

## 9.65.2 Member Function Documentation

### 9.65.2.1 getChannel()

```
channel_t Experimental::TransformGaussianBroadening::getChannel ( )
```

### 9.65.2.2 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:

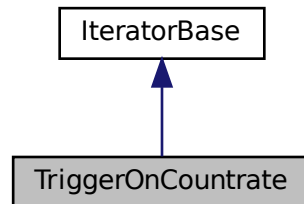
- [Iterators.h](#)

## 9.66 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.*

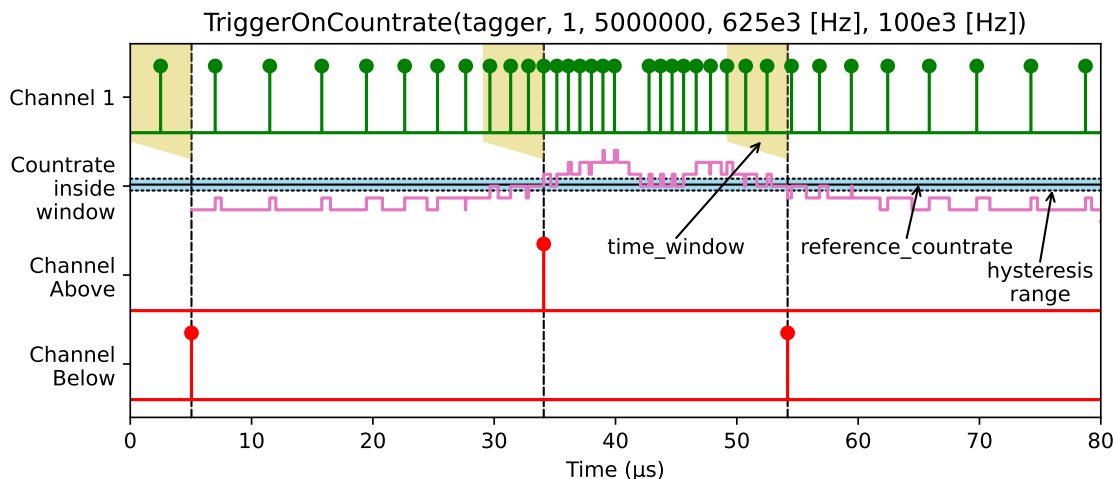
## Protected Member Functions

- `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*

## Additional Inherited Members

### 9.66.1 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`.

### 9.66.2 Constructor & Destructor Documentation

#### 9.66.2.1 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 <a href="#">TimeTagger</a> object.
<i>input_channel</i>	Channel number of the channel whose count rate will control the trigger channels.
<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.

## 9.66.2.2 ~TriggerOnCountrate()

```
TriggerOnCountrate::~~TriggerOnCountrate ( )
```

## 9.66.3 Member Function Documentation

## 9.66.3.1 getChannelAbove()

```
channel_t TriggerOnCountrate::getChannelAbove ( )
```

Get the channel number of the `above` channel.

## 9.66.3.2 getChannelBelow()

```
channel_t TriggerOnCountrate::getChannelBelow ( )
```

Get the channel number of the `below` channel.

## 9.66.3.3 getChannels()

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

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



#### 9.66.3.4 getCurrentCountrate()

```
double TriggerOnCountrate::getCurrentCountrate ( )
```

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

#### 9.66.3.5 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.

#### 9.66.3.6 isAbove()

```
bool TriggerOnCountrate::isAbove ( )
```

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

#### 9.66.3.7 isBelow()

```
bool TriggerOnCountrate::isBelow ( )
```

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

#### 9.66.3.8 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](#).

**9.66.3.9 on\_start()**

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

callback when the measurement class is started

This function is guarded by the update lock.

Reimplemented from [IteratorBase](#).

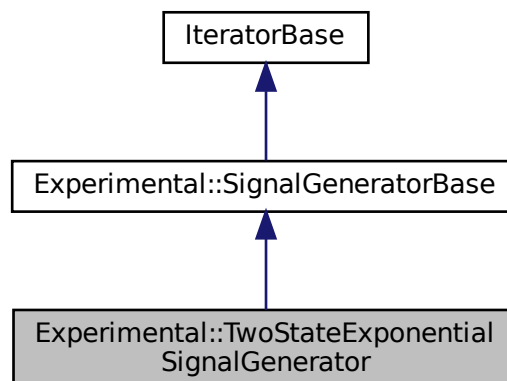
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.67 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](#) ()

## Protected Member Functions

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

## Additional Inherited Members

### 9.67.1 Constructor & Destructor Documentation

#### 9.67.1.1 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.

#### 9.67.1.2 ~TwoStateExponentialSignalGenerator()

```
Experimental::TwoStateExponentialSignalGenerator::~~TwoStateExponentialSignalGenerator ( )
```

### 9.67.2 Member Function Documentation

### 9.67.2.1 `get_next()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.67.2.2 `initialize()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.67.2.3 `on_restart()`

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

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

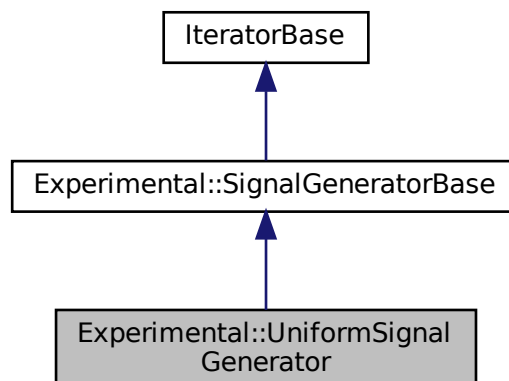
The documentation for this class was generated from the following file:

- [Iterators.h](#)

## 9.68 `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](#) ()

## Protected Member Functions

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

## Additional Inherited Members

### 9.68.1 Constructor & Destructor Documentation

#### 9.68.1.1 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.

#### 9.68.1.2 ~UniformSignalGenerator()

```
Experimental::UniformSignalGenerator::~~UniformSignalGenerator ( )
```

### 9.68.2 Member Function Documentation

### 9.68.2.1 `get_next()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.68.2.2 `initialize()`

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

Implements [Experimental::SignalGeneratorBase](#).

### 9.68.2.3 `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](#)

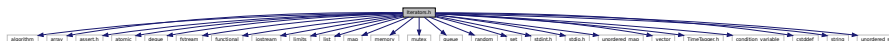
## Chapter 10

# File Documentation

### 10.1 Iterators.h File Reference

```
#include <algorithm>
#include <array>
#include <assert.h>
#include <atomic>
#include <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 [TimeDifferencesImpl< T >](#)
- 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 [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 raising 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 [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::SignalGeneratorBase](#)
- class [Experimental::UniformSignalGenerator](#)
- class [Experimental::GaussianSignalGenerator](#)
- class [Experimental::TwoStateExponentialSignalGenerator](#)
- class [Experimental::MarkovProcessGenerator](#)
- class [Experimental::ExponentialSignalGenerator](#)
- class [Experimental::GammaSignalGenerator](#)
- class [Experimental::PoissonSignalGenerator](#)
- class [Experimental::PatternSignalGenerator](#)
- class [Experimental::SimSignalSplitter](#)
- class [Experimental::TransformEfficiency](#)
- class [Experimental::TransformGaussianBroadening](#)
- class [Experimental::TransformDeadtime](#)
- class [Experimental::TransformCrosstalk](#)
- class [Experimental::SimDetector](#)
- class [Experimental::SimLifetime](#)

## Namespaces

- [Experimental](#)

## Macros

- `#define BINNING\_TEMPLATE\_HELPER(fun_name, binner, ...)`  
*[FastBinning](#) caller helper.*

## Enumerations

- enum [CoincidenceTimestamp](#) : uint32\_t { [CoincidenceTimestamp::Last](#) = 0, [CoincidenceTimestamp::Average](#) = 1, [CoincidenceTimestamp::First](#) = 2, [CoincidenceTimestamp::ListedFirst](#) = 3 }  
*type of timestamp for the [Coincidence](#) virtual channel (Last, Average, First, ListedFirst)*
- enum [GatedChannelInitial](#) : uint32\_t { [GatedChannelInitial::Closed](#) = 0, [GatedChannelInitial::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 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 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](#).

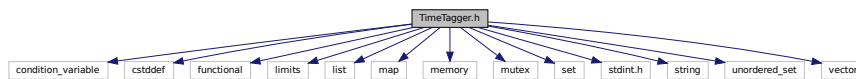
#### Enumerator

UNKNOWN	
HIGH	
LOW	

## 10.2 TimeTagger.h File Reference

```
#include <condition_variable>
#include <cstdint>
#include <functional>
#include <limits>
#include <list>
#include <map>
#include <memory>
#include <mutex>
#include <set>
#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 TimeTagger based on dump files*
- class [TimeTaggerNetwork](#)  
*network TimeTagger 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.*

## 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](#) "2.14.0"  
*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 [Resolution](#) { [Resolution::Standard](#) = 0, [Resolution::HighResA](#) = 1, [Resolution::HighResB](#) = 2, [Resolution::HighResC](#) = 3 }
- This enum selects the high resolution mode of the Time Tagger series.*
- enum [ChannelEdge](#) : int32\_t {  
[ChannelEdge::NoFalling](#) = 1 << 0, [ChannelEdge::NoRising](#) = 1 << 1, [ChannelEdge::NoStandard](#) = 1 << 2, [ChannelEdge::NoHighRes](#) = 1 << 3,  
[ChannelEdge::All](#) = 0, [ChannelEdge::Rising](#) = 1, [ChannelEdge::Falling](#) = 2, [ChannelEdge::HighResAll](#) = 4,  
[ChannelEdge::HighResRising](#) = 4 | 1, [ChannelEdge::HighResFalling](#) = 4 | 2, [ChannelEdge::StandardAll](#) = 8,  
[ChannelEdge::StandardRising](#) = 8 | 1,  
[ChannelEdge::StandardFalling](#) = 8 | 2 }
- Enum for filtering the channel list returned by getChannelList.*
- enum [FpgaLinkInterface](#) { [FpgaLinkInterface::SFPP\\_10GE](#) }

*Enum for selecting the fpga link output interface.*

- enum `LogLevel` { `LOGGER_ERROR` = 40, `LOGGER_WARNING` = 30, `LOGGER_INFO` = 10 }
- enum `AccessMode` { `AccessMode::Listen` = 0, `AccessMode::Control` = 2, `AccessMode::SynchronousControl` = 3 }
- enum `LanguageUsed` : `std::uint32_t` {  
`LanguageUsed::Cpp` = 0, `LanguageUsed::Python`, `LanguageUsed::Csharp`, `LanguageUsed::Matlab`,  
`LanguageUsed::Labview`, `LanguageUsed::Mathematica`, `LanguageUsed::Unknown` = 255 }
- enum `FrontendType` : `std::uint32_t` {  
`FrontendType::Undefined` = 0, `FrontendType::WebApp`, `FrontendType::Firefly`, `FrontendType::Pyro5RPC`,  
`FrontendType::UserFrontend` }
- enum `UsageStatisticsStatus` { `UsageStatisticsStatus::Disabled`, `UsageStatisticsStatus::Collecting`, `UsageStatisticsStatus::Collected` }

## 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 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 [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](#)

## 10.2.1 Macro Definition Documentation

### 10.2.1.1 channel\_t

```
#define channel_t int
```

The type for storing a channel identifier.

### 10.2.1.2 ErrorLog

```
#define ErrorLog(
    ... ) LogMessage (LOGGER_ERROR, __VA_ARGS__);
```

### 10.2.1.3 ErrorLogSuppressed

```
#define ErrorLogSuppressed(
    ... ) LogMessageSuppressed (LOGGER_ERROR, __VA_ARGS__);
```

#### 10.2.1.4 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.

#### 10.2.1.5 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)
```

#### 10.2.1.6 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)
```

#### 10.2.1.7 GET\_DATA\_2D

```
#define GET_DATA_2D(
    function_name,
    type,
    argout,
    attribute ) attribute void function_name(std::function<type *(size_t, size_t)>
argout)
```



### 10.2.1.8 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)
```

### 10.2.1.9 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)
```

### 10.2.1.10 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)
```

### 10.2.1.11 InfoLog

```
#define InfoLog(
    ... ) LogMessage(LOGGER\_INFO, __VA_ARGS__);
```

### 10.2.1.12 InfoLogSuppressed

```
#define InfoLogSuppressed(  
    ... ) LogMessageSuppressed(LOGGER\_INFO, __VA_ARGS__);
```

### 10.2.1.13 LogMessage

```
#define LogMessage(  
    level,  
    ... ) LogBase(level, __FILE__, __LINE__, false, __VA_ARGS__);
```

### 10.2.1.14 LogMessageSuppressed

```
#define LogMessageSuppressed(  
    level,  
    ... ) LogBase(level, __FILE__, __LINE__, true, __VA_ARGS__);
```

### 10.2.1.15 timestamp\_t

```
#define timestamp_t long long
```

The type for all timestamps used in the Time Tagger suite, always in picoseconds.

### 10.2.1.16 TIMETAGGER\_VERSION

```
#define TIMETAGGER_VERSION "2.14.0"
```

The version of this software suite.

### 10.2.1.17 TT\_API

```
#define TT_API __declspec(dllimport)
```

### 10.2.1.18 WarningLog

```
#define WarningLog(  
    ... ) LogMessage(LOGGER\_WARNING, __VA_ARGS__);
```

### 10.2.1.19 WarningLogSuppressed

```
#define WarningLogSuppressed(  
    ... ) LogMessageSuppressed(LOGGER\_WARNING, __VA_ARGS__);
```

## 10.2.2 Typedef Documentation

### 10.2.2.1 \_Iterator

```
using \_Iterator = IteratorBase
```

### 10.2.2.2 logger\_callback

```
typedef void(* logger_callback) (LogLevel level, std::string msg)
```

## 10.2.3 Enumeration Type Documentation

### 10.2.3.1 AccessMode

```
enum AccessMode [strong]
```

Enumerator

<a href="#">Listen</a>	
<a href="#">Control</a>	
<a href="#">SynchronousControl</a>	

### 10.2.3.2 ChannelEdge

```
enum 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	

### 10.2.3.3 FpgaLinkInterface

```
enum FpgaLinkInterface [strong]
```

Enum for selecting the fpga link output interface.

#### Enumerator

SFPP_10GE	
-----------	--

### 10.2.3.4 FrontendType

```
enum FrontendType : std::uint32_t [strong]
```

#### Enumerator

Undefined	
WebApp	
Firefly	
Pyro5RPC	
UserFrontend	

### 10.2.3.5 LanguageUsed

```
enum LanguageUsed : std::uint32_t [strong]
```

#### Enumerator

Cpp	
Python	
Csharp	
Matlab	
Labview	
Mathematica	
Unknown	

### 10.2.3.6 LogLevel

```
enum LogLevel
```

#### Enumerator

LOGGER_ERROR	
LOGGER_WARNING	
LOGGER_INFO	

### 10.2.3.7 Resolution

```
enum 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	

## 10.2.3.8 UsageStatisticsStatus

```
enum UsageStatisticsStatus [strong]
```

## Enumerator

Disabled	
Collecting	
CollectingAndUploading	

## 10.2.4 Function Documentation

## 10.2.4.1 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

## 10.2.4.2 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.
----------------	--

### 10.2.4.3 createTimeTaggerVirtual()

```
TT_API TimeTaggerVirtual* createTimeTaggerVirtual ( )
```

default constructor factory for the createTimeTaggerVirtual class.

### 10.2.4.4 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

### 10.2.4.5 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

#### 10.2.4.6 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
---------------	--

#### 10.2.4.7 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.

#### 10.2.4.8 getTimeTaggerModel()

```
TT_API std::string getTimeTaggerModel (
    const std::string & serial )
```

#### 10.2.4.9 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.
----------------	---

#### 10.2.4.10 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.

#### 10.2.4.11 `getUsageStatisticsStatus()`

```
TT_API UsageStatisticsStatus getUsageStatisticsStatus ( )
```

gets the status of the usage statistics system.

#### Returns

the current status of the usage statistics system.

#### 10.2.4.12 `getVersion()`

```
TT_API std::string getVersion ( )
```

Get the version of the [TimeTagger](#) cxx backend.

#### 10.2.4.13 `hasTimeTaggerVirtualLicense()`

```
TT_API bool hasTimeTaggerVirtualLicense ( )
```

Check if a license for the [TimeTaggerVirtual](#) is available.

#### 10.2.4.14 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.

#### 10.2.4.15 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 ' <a href="#">FileWriter</a> ', 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.

#### 10.2.4.16 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.

#### 10.2.4.17 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.

#### 10.2.4.18 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.
--------------------	-------------------------------------

#### 10.2.4.19 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.
-----------------	------------------------------------

#### 10.2.4.20 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.

**10.2.4.21 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

**10.2.4.22 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>
---------------	--

### 10.2.4.23 `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

<code>new_status</code>	new status of the usage statistics system.
-------------------------	--

## 10.2.5 Variable Documentation

### 10.2.5.1 `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.

### 10.2.5.2 `CHANNEL_UNUSED_OLD`

```
constexpr channel_t CHANNEL_UNUSED_OLD = -1 [constexpr]
```

### 10.2.5.3 `TT_CHANNEL_FALLING_EDGES`

```
constexpr ChannelEdge TT_CHANNEL_FALLING_EDGES = ChannelEdge::Falling [constexpr]
```

### 10.2.5.4 `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.

#### 10.2.5.5 TT\_CHANNEL\_NUMBER\_SCHEME\_ONE

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_ONE = 2 [constexpr]
```

#### 10.2.5.6 TT\_CHANNEL\_NUMBER\_SCHEME\_ZERO

```
constexpr int TT_CHANNEL_NUMBER_SCHEME_ZERO = 1 [constexpr]
```

#### 10.2.5.7 TT\_CHANNEL\_RISING\_AND\_FALLING\_EDGES

```
constexpr ChannelEdge TT_CHANNEL_RISING_AND_FALLING_EDGES = ChannelEdge::All [constexpr]
```

#### 10.2.5.8 TT\_CHANNEL\_RISING\_EDGES

```
constexpr ChannelEdge TT_CHANNEL_RISING_EDGES = ChannelEdge::Rising [constexpr]
```