TCSPC Counter Operating Instructions_V3_EN.docx

Version: 3.0

Six-Channel TCSPC Operating Instructions

Table of Contents

1 Quick Setup

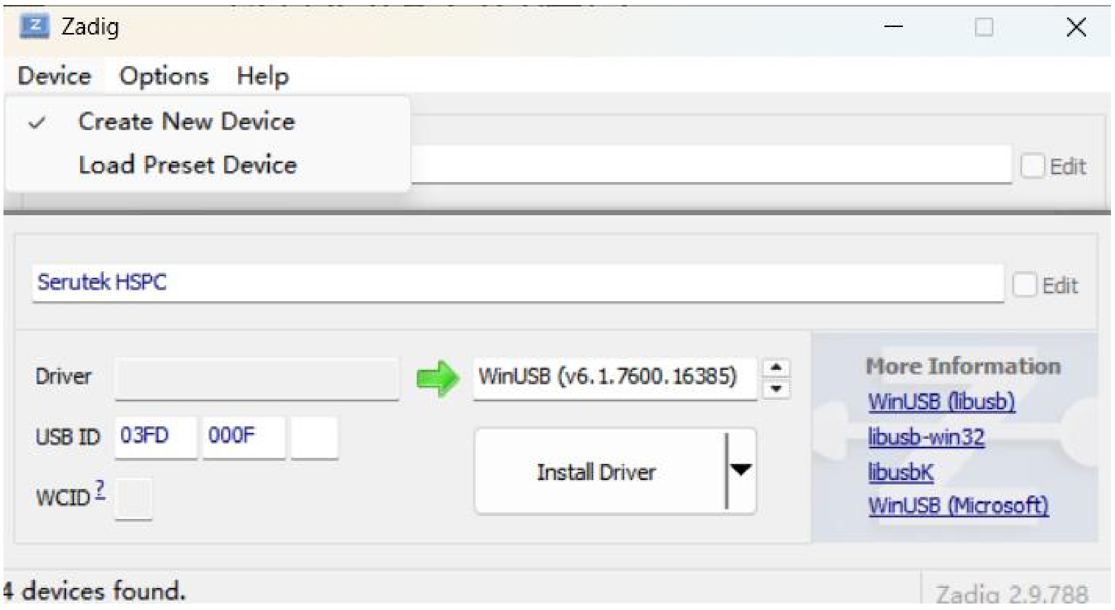
1.1 Host Computer Configuration

It is recommended to use a host computer with Windows 11 operating system, with memory of 8GB or above, and preferably equipped with an SSD. If using a laptop, it is recommended to operate with the power adapter connected.

1.2 Install USB Driver

Use Zadig-2.9 to install the driver. Zadig is included in the installation package. Do not connect the counter first. Double-click to run Zadig on the host computer.

Click on the menu bar: Device -> Create New Device. Enter "Serutek HSPC" in the Name field. Enter "03FD 000F" in the USB ID field. Then click "Install Driver".



Serutek HSPC

Driver → WinUSB (v6.1.7600.16385)

USB ID 03FD 000F

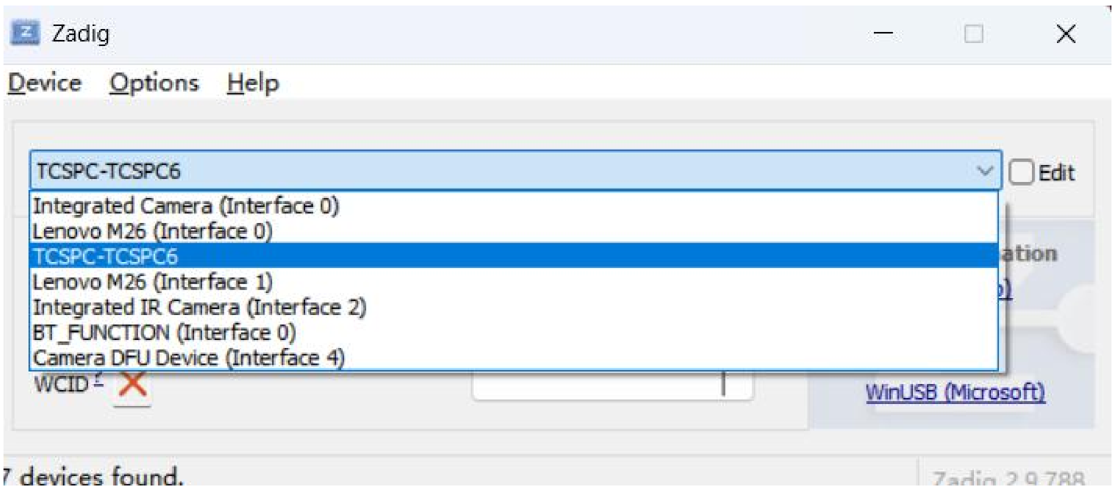
WCID ?

4 devices found.

Zadig 2.9.788

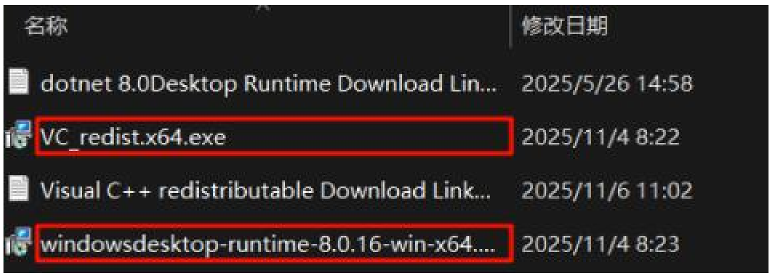
Wait for the installation to complete. Power on the counter. After 2 seconds, connect the counter to the host computer using a USB 3.0 male-to-male cable. **Pay special attention to connect to a USB 3.0 or higher port on the host computer.** If connected to a USB 2.0 port, the system will not work properly. If the driver installation is successful, the host computer will emit a prompt sound, and you will see the "SIMTURM-TCHPC6" device in the USB devices list at the bottom right corner of the taskbar. If the device is not seen, power cycle the counter 1-2 times. If repeated attempts fail, please contact our company, and an engineer will assist you remotely.

In the Zadig software, please confirm that the driver version for the "SIMTURM-TCHPC6" device matches the one shown in the figure below. If not, click "Upgrade Driver" or reinstall the driver.



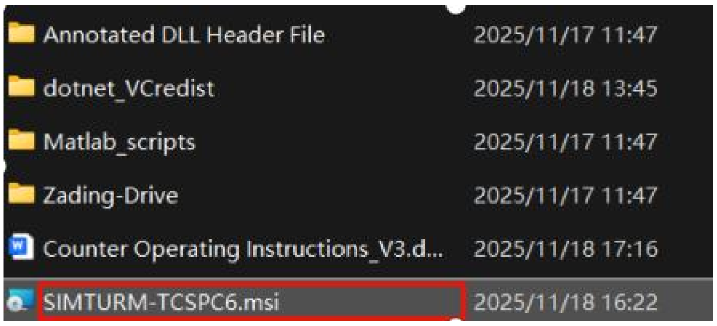
1.3 Install Host Computer Program

Before installing the host computer software, you need to install the latest version of Microsoft Visual C++ Redistributable and DotNet 8.0 Desktop Runtime. Please refer to the links provided in the installation package.



名称	修改日期
dotnet 8.0Desktop Runtime Download Lin...	2025/5/26 14:58
VC_redist.x64.exe	2025/11/4 8:22
Visual C++ redistributable Download Link...	2025/11/6 11:02
windowsdesktop-runtime-8.0.16-win-x64....	2025/11/4 8:23

Install the program package "SIMTURM-TCHPC6.msi". Double-click the installer and follow the prompts.



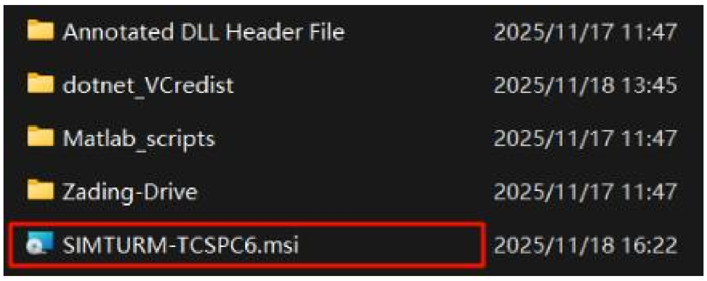
Annotated DLL Header File	2025/11/17 11:47
dotnet_VCredist	2025/11/18 13:45
Matlab_scripts	2025/11/17 11:47
Zading-Drive	2025/11/17 11:47
Counter Operating Instructions_V3.d...	2025/11/18 17:16
SIMTURM-TCSPC6.msi	2025/11/18 16:22

After installation, you can click the shortcut "SIMTURM-TCHPC6" on the desktop to launch the host software:

1.4 Uninstall Host Computer Software

If you need to upgrade the host software, you must first uninstall the old version. The uninstallation method is as follows:

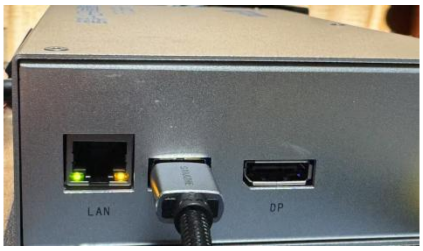
Click on SIMTURM-TCSPC6.msi -> Uninstall SIMTURM-TCHPC6, as shown below.



Annotated DLL Header File	2025/11/17 11:47
dotnet_VCredist	2025/11/18 13:45
Matlab_scripts	2025/11/17 11:47
Zading-Drive	2025/11/17 11:47
SIMTURM-TCSPC6.msi	2025/11/18 16:22

1.5 Connect Cables

1. Power on the counter. After the counter initialization is complete, the yellow LED on the LAN port lights up. If the yellow LED flashes, it indicates that the counter initialization failed, possibly due to unauthorized firmware, internal clock not locked, or other internal exceptions.
2. Power on the host computer.
3. Connect the counter to the host computer using a USB 3.0 male-to-male cable. **It is crucial to connect to a USB 3.0 port on the computer.** If connected to a USB 2.0 port, the system will not function properly. When the USB connection is normal, the green LED on the LAN port lights up.



4. Start the host software SRPC (Se Ru Photon Counter). SRPC will detect if the counter is connected upon startup. If the software can find the device, the status bar at the bottom of the software interface will display "USB Connected". Otherwise, it will show "! USB Not Connected", and the software interface will be inoperable. The USB connection is hot-pluggable when the counter is not performing measurements. If the USB connection is disconnected while the counter is measuring, it may cause internal errors in the counter, requiring a restart of the counter.

2 Working Principle and Functions

2.1 Working Principle

The TCSPC single photon counter operates based on TDC (Time-to-Digital Conversion), recording the timestamp corresponding to input events. Each input channel is equipped with a TDC. The rising edge of the voltage pulse signal input to that channel triggers the TDC recording action, which records and outputs the timestamp corresponding to that rising edge.

The TDCs on each channel operate independently but remain synchronized in time. When the counter receives the start measurement command, the TDC internal logic starts counting. This moment is the origin (or zero point) of the timestamps. The TDC will roll over due to overflow after counting for a certain period. The current version's rollover period is approximately 52,000 seconds[^1], about 14 hours. When the measurement stops, the timestamp resets. In most cases, a single measurement does not last as long as 14 hours, so users do not need to worry about timestamp overflow. If truly long-duration single measurements are required, compensation for overflowed timestamps is possible. For specific methods, please contact our company.

The host computer communicates with the counter via USB 3.0. All measurement functions: timestamp, time difference, histogram, and intensity measurement can be completed via USB 3.0. A 10 Gigabit optical port can be used as the output port for timestamps (T2) and time differences (T3). When using the 10 Gigabit Ethernet port as the data output port, it can continuously output a timestamp data stream at a maximum rate of 200 Msps.

Outputting a timestamp stream is the basic function of this counter. Other functions, such as time difference output and histogram output, are derived from processing the timestamps.

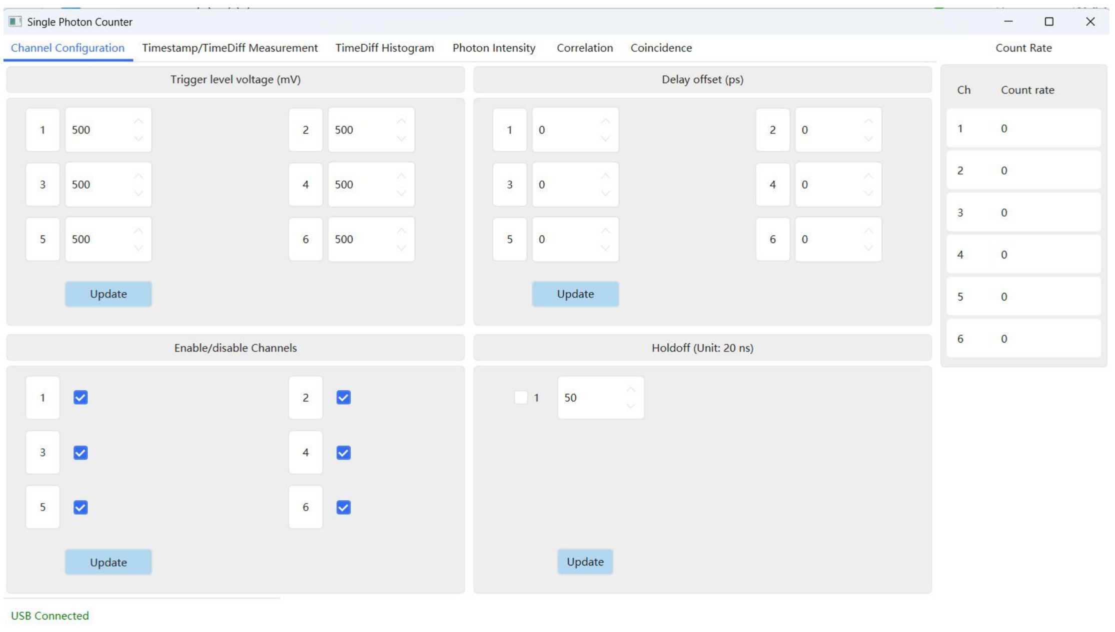
2.2 Functional Overview

2.2.1 Channel Quantity and Function

This counter has 6 independent channels. In the standard version, channels 1-4 are one group, and channels 5-6 are another group. Each group has an output capability of 100 Msps. In practical testing, channel 5 or 6 can be used for high-frequency sync signals and event identification, while the remaining channels are used to measure single-photon pulses output by detectors.

In the L version, data from channels 1-6 are aggregated and output via USB 3.0, with a saturated count rate of 39 Msps.

On the "Parameter Settings" page, you can set the trigger voltage threshold and delay compensation for each channel. You can also disable or enable specific channels. Additionally, a hold-off function can be set for channel 1.



Trigger level voltage (mV): 1 500, 2 500, 3 500, 4 500, 5 500, 6 500

Delay offset (ps): 1 0, 2 0, 3 0, 4 0, 5 0, 6 0

Enable/disable Channels: 1 ✓, 2 ✓, 3 ✓, 4 ✓, 5 ✓, 6 ✓

Holdoff (Unit: 20 ns): ☐ 1 50

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

USB Connected

2.2.2 Channel Delay Compensation

The delay can be adjusted independently for each specific channel. The adjustment range is approximately the range of the Int32 integer type, with a step of 1 ps. **It is important to note that this adjustment only adds a compensation value to the timestamp; it does not physically advance or delay the signal arrival time.** Therefore, in time difference output mode and real-time histogram mode, it is recommended to keep the adjustment amount smaller than the period of the gate (sync) signal. For example: if the gate signal period is 10 MHz, corresponding to a signal period of 100 ns, then the channel delay adjustment should be less than 100 ns to ensure the correct operation of the internal time difference calculation module.

2.2.3 Channel Trigger Voltage Setting

Each channel can accept electrical signals with a maximum voltage of 5V, with an input impedance of 50 ohms. Each channel is equipped with a comparator. The comparator's threshold voltage is independently adjustable, with a range of 0mV to 4096mV and a step of 1mV. When the input signal voltage exceeds the set threshold voltage, the comparator output transitions from low to high level, creating a rising edge, which is then recorded by the TDC.

2.2.4 Channel Enable

You can independently disable/enable specific channels. **Note that enabling/disabling here does not affect the count rate display, but it does affect functions related to timestamp/time difference measurement.** After disabling a specific channel, that channel's measurement data will not appear in recorded files or be displayed in histograms.

2.2.5 Hold-off Function for Channel 1

You can choose to enable or disable the trigger hold-off function for channel 1. When enabled, you can also adjust the hold-off time. The hold-off time setting value ranges from 0 to 127, with a unit of approximately 20 ns. The maximum settable hold-off time is about 2.5 us. During the hold-off period, the trigger function is disabled. Even if pulses meeting the trigger condition arrive, channel 1 will not trigger, effectively increasing the dead time of channel 1.

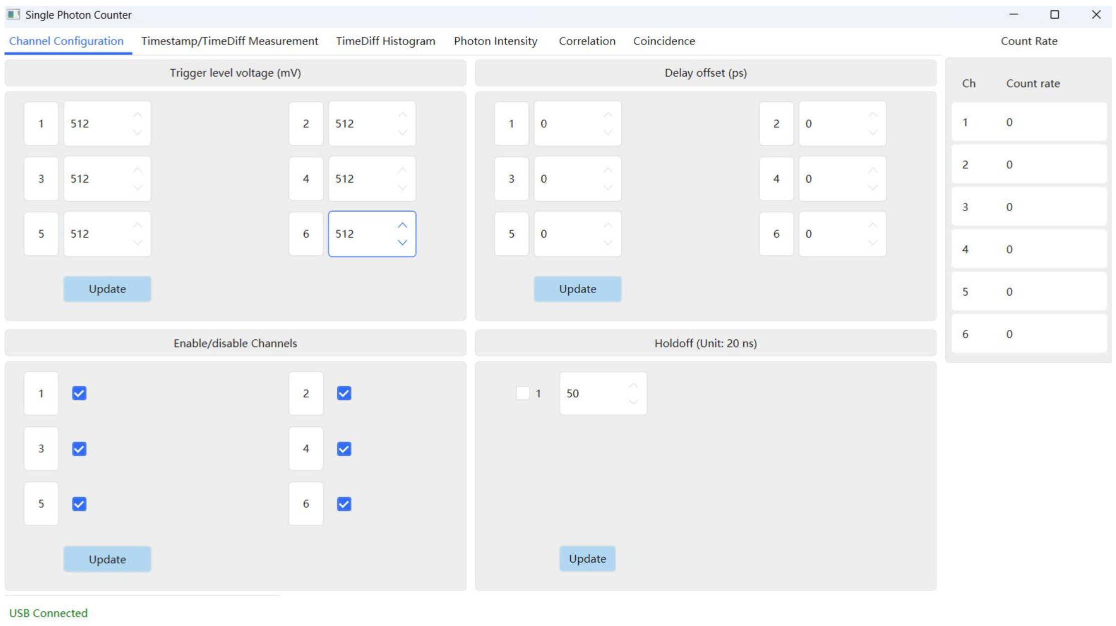
The trigger hold-off function can be used in the following two scenarios:

When the sync channel is connected to channel 1 and the count rate is high. By setting a larger hold-off time, the actual count rate of the sync signal can be reduced, thereby saving data bandwidth.

When the sync signal has a slow falling edge, or the signal quality is poor with significant noise, using the hold-off function allows the hold-off time to cover the entire falling edge, thus avoiding false triggering.

2.2.6 Count Rate Display

After the host computer successfully establishes communication with the counter, the count rate for each channel will be displayed in real-time. The count rate updates once per second.



Trigger level voltage (mV)

1	512	2	512
3	512	4	512
5	512	6	512

Delay offset (ps)

1	0	2	0
3	0	4	0
5	0	6	0

Enable/disable Channels

1	✓	2	✓
3	✓	4	✓
5	✓	6	✓

Holdoff (Unit: 20 ns)

☐	1	50

Count Rate

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

2.2.7 Timestamp Output Mode (T2 Mode)

In this mode, the single photon counter sends the raw timestamp measurement data to the host computer in real-time via USB 3.0 or the two 10 Gigabit SFP+ optical ports (standard version). The host computer initiates a synchronized reception function and saves the received timestamps as a binary file. When using the 10 Gigabit Ethernet port for data transfer, each measurement will save two binary files: one records data from channels 1-4, and the other records data from channels 5-6.

The binary file format is: every 8 bytes corresponds to one timestamp. Therefore, one timestamp is 64 bits, defined as follows:

Bits [63:57]	**Bits [56:0]**
Unsigned integer, Channel Number	Timestamp, signed integer, LSB unit: ps

For specific parsing methods, please refer to the provided Matlab scripts.

2.2.8 Time Difference Output Mode (T3 Mode)

In T3 mode, the counter outputs the time difference of each channel relative to the sync signal, while simultaneously recording the timestamp of the corresponding sync signal arrival, forming the so-called time-tagged time-resolved stream (TTTR information stream) output to the host computer, saved as a binary file. During measurement, the user needs to select the sync signal channel as the start signal (gate open), and the other channels as stop signals (gate close). The output is the time lag of the stop signal relative to the start signal.

In this mode, the user needs to select the gate open signal channel (Start). After the measurement starts, the counter internally calculates the time difference of the other channels relative to the sync channel based on timestamps and simultaneously records the sync timestamp. If no photon arrives within a sync signal cycle, that sync signal's timestamp is not recorded.

T3 mode can output via USB 3.0 or the 10 Gigabit Ethernet port. When using USB 3.0, the maximum count rate is 39 Msps. When using the 10 Gigabit Ethernet port, the 2nd port is used, with a maximum count rate of 100 Msps (when using the 10 Gigabit port, the host software creates two binary files: xxx1.bin, xxx2.bin. In T3 mode, TTTR information is saved in the second file).

The TTTR binary file has a 64-bit record for each entry. The upper 7 bits indicate the channel number, and the remaining bits have different meanings depending on whether it is a sync signal. If the record is a sync signal, bits [56:0] represent the timestamp of the sync signal. If it is not a sync signal, bits [56:0] represent the time difference of that channel relative to the sync signal.

When the channel is the sync signal:

Bits [63:57]	**Bits [56:0]**
Unsigned integer, Channel Number	Timestamp, signed integer, unit: ps

When the channel is not the sync signal:

Parsing the time difference file, each 8 bytes represent a time difference record, format as follows:

Bits [63:57]	**Bits [56:0]**
Unsigned integer, Channel Number	Time difference relative to sync, ps

From the time difference data format, it can be seen that the representable time difference range is ±2^56 ps, approximately 40 hours, sufficient for most applications.

The following data is excerpted from a TTTR parsed text file:

```text
6:  197969

2:  2215

5:  2790

1:  2017

3:  2406

4:  2605

6:  364643

2:  2185

5:  2791

1:  2000

3:  2394

4:  2586
```

In this measurement, channel 6 is the sync signal, and channels 1,2,3,4,5 are photon detection channels. Records for channel 6 represent sync timestamps, and records for other channels represent photon time differences relative to that sync.

2.2.9 Real-time Histogram

The real-time histogram mode utilizes the T2 or T3 data output by the counter to achieve high-speed time difference histogram statistics. The user needs to set one channel as the gate open signal and select the gate close signal(s) for histogram plotting (multiple selections). After the measurement starts, real-time histograms of the time differences of multiple channels relative to the sync channel can be displayed simultaneously.

The user can select the histogram bin size and number on the interface. When the sync signal frequency is high (>20MHz), it is recommended to use channel 5 or 6 as the sync signal.

On the histogram interface, the user needs to set the following parameters:

1. **Statistics Mode**, Timestamp (T2) or Time Difference (T3). This mode determines whether the counter outputs T2 or T3 data. When T2 data is selected, the host software calculates the time difference of each channel relative to the sync channel based on the timestamps output by the counter. When T3 data is selected, the counter directly outputs the internally calculated time difference data, and the host software performs histogram statistics based on the time difference data.

In most applications, the results obtained from T2 and T3 modes are the same. However, when a large number of photons arrive within each sync pulse interval (e.g., thousands or tens of thousands of photons), and the counter's internal T3 module cannot handle it correctly (the TCSPC module can handle about 300 photons/sync cycle, the HSPC standard board about 5000 photons/sync cycle), the Timestamp mode (T2) can be used.

2. **Refresh Interval**, unit milliseconds. Minimum value is 500 (ms). This parameter determines how often the histogram is calculated. When the count rate is high (>20M), it is recommended to choose a refresh interval less than or equal to 1000 ms.

3. **Scaling**, magnification factor, range 0~40. The current version uses 64 bits to represent time difference, with sufficient range. Therefore, scaling is unnecessary; set it to 0.

4. **Gate Open Channel**, selects the gate open channel for time difference calculation, i.e., the start signal.

5. **Gate Close Channel(s)**, i.e., the stop signal(s) for time difference calculation. Multiple gate close channels can be selected simultaneously. Changing gate close channels requires restarting the measurement.

6. **Update Mode**. Two modes: Accumulate and Refresh. In Accumulate mode, the count values read each time are added to the previous count values for that bin. In Refresh mode, the currently read count values are displayed, working like an oscilloscope; some devices also call this Oscilloscope mode.

7. **Histogram Start Point**, unit ps, default 0. When the histogram peak is far from the sync signal (large time difference), you can set the histogram start point or offset to reduce statistical calculations for useless regions.

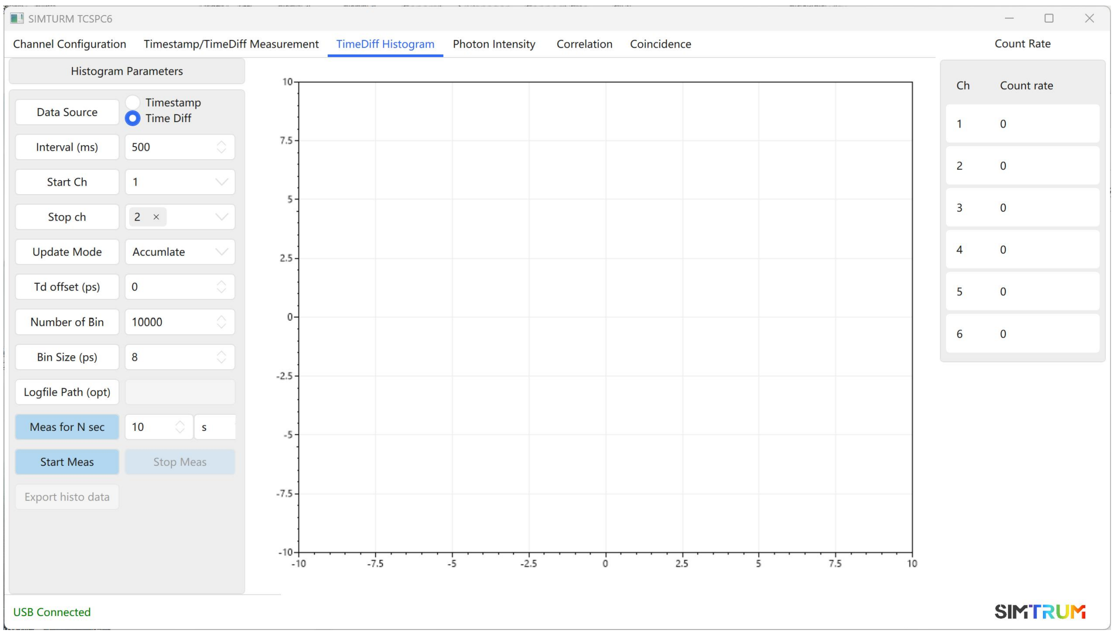
8. **Bin Count**. The bin count, bin size(LSB), and scale together determine the histogram range. Histogram Range = 2^scale * bin size * bin number (ps). For example:

Scale = 0; bin size = 10; bin number = 10000, then Histogram Range = 2^0 * 10 * 10000 = 100000 ps = 100 ns.
Although theoretically the bin count is unlimited, a larger bin count increases the computational and display load on the host computer. It is recommended that in Timestamp mode, bin count ≤ 1e6; in Time Difference mode, bin count ≤ 1e7.

9. **Bin Size(LSB)**, the actual width of each histogram bin = 2^scale * bin size(LSB).

10. **Record Historical Data**. Click this button to select a save path. After measurement starts, histogram data will be recorded in a file in binary format. A Matlab script is provided to parse this binary file.

11. **Export Histogram Data**. After measurement ends, click this button to save the histogram data from the last refresh cycle to a user-specified file. The save format is a text file. When using this function, it is recommended to select Accumulate as the update mode.



Data Source: Timestamp [✓ Time Diff]
Interval (ms): 500
Start Ch: 1
Stop ch: 2
Update Mode: Accumlate
Td offset (ps): 0
Number of Bin: 10000
Bin Size (ps): 8
Logfile Path (opt):
Meas for N sec: 10 s

Count Rate

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

2.2.10 Photon Counting Statistics

Photon counting statistics refer to counting the number of photons on a channel over a period of time. Two modes can be selected during measurement.

Time Window Mode. In this mode, the user needs to specify the statistical window length, unit is us. The counter measures the photon count on each channel within the specified time interval.
Sync Channel Trigger Mode. In this mode, the user needs to designate the sync signal channel number. The counter measures the number of photons arriving on each channel between two sync signals.

After measurement starts, the host software reads the count values output by the counter at a frequency of once per second and plots the curve in real-time. The user can also choose to save the counting results in a binary file for post-measurement analysis.

The photon intensity binary file format is as follows. Each record is 64 bits (8 bytes). The first record (8 bytes) is the file header, recording measurement information such as sync channel and statistical window length. Subsequently, every 64 bytes form a group of data, where each 8 bytes represents the count value of a channel, from channel 1 to channel 6 sequentially. When measuring in Time Window mode, this group of data represents the counts of 6 channels within one window.

When measuring in Sync Signal mode, this group of data represents the counts of 6 channels between one sync pulse and the next sync pulse.

The format of each record is as follows:

Bits [63:40]	**Bits [39:32]**	**Bits [31:0]**
Unsigned int	Unsigned int	Unsigned int
Sequence number of the sync pulse corresponding to the count value	Channel Number	Count Value

In Time Window mode, the bits **[63:40]** of the record are meaningless and can be ignored. Bits [39:32] (8 bits) indicate the channel number, and the lower 32 bits of the record represent the count value. For example: 0x00000302000002A0 indicates the count value for channel 2 is 672 (0x2A0). This count value represents the photon count on channel 2 between the 3rd and 4th pulses of the sync channel.

In Sync Signal mode, bits **[63:40]** of the record represent the sequence number of the sync pulse corresponding to the count value.

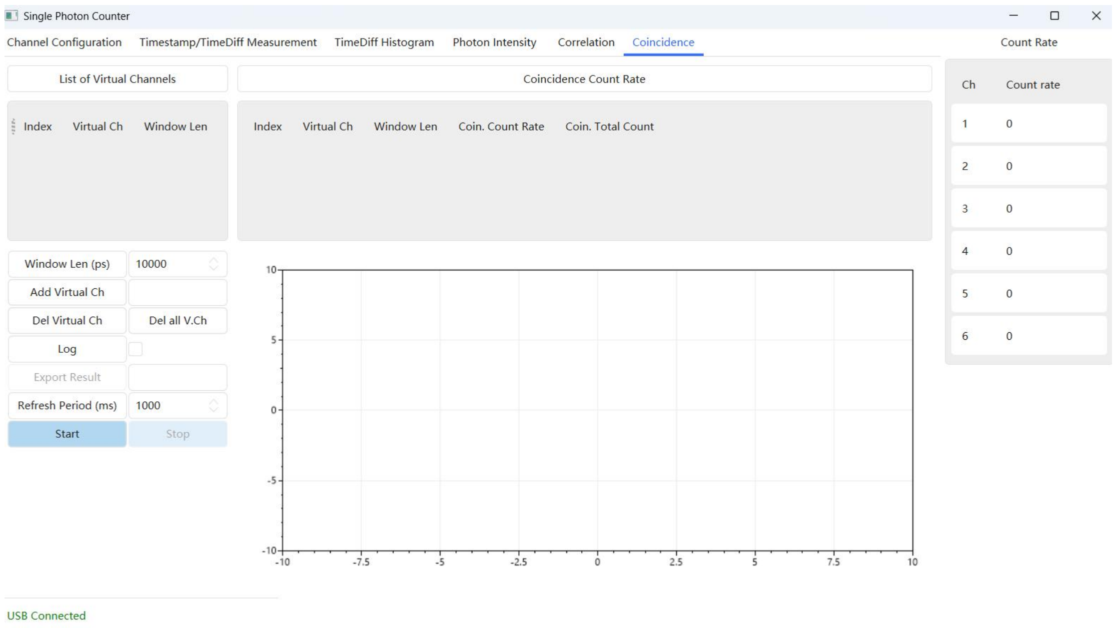
A Matlab script file is provided to parse the recorded photon intensity binary file, storing the parsed results in the workspace.

The header format is as follows:

Bits [63:40]	**Bits [39:32]**	**Bits [31:0]**
Sync signal channel number (sync)	0x00	Time window length (us) (intense_window)

When sync == 0, it indicates measurement is in Time Window mode. In this case, the 4th byte of the header represents the time window length, unit us. When sync == 1~6, it indicates measurement is in Sync Pulse mode, with the sync signal channel being the value of sync. In this case, the lower 32-bit value can be ignored.

2.2.11 Coincidence Counting



Index	Virtual Ch	Window Len

Index	Virtual Ch	Window Len	Coin. Count Rate	Coin. Total Count

Window Len (ps) 10000

Refresh Period (ms) 1000

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

This unit provides powerful and flexible coincidence counting functions. Users can set multiple coincidence strategies. Each strategy can independently set the channels involved in coincidence calculation and the coincidence window size. There is no hard limit on the number of strategies, limited only by the host computer's computational capability. With a CPU with a higher number of cores, implementing simultaneous coincidence counting for hundreds of strategies is easily achievable.

In this counter, one strategy corresponds to one virtual channel. A virtual channel needs to specify the channels participating in coincidence calculation and the coincidence window width (unit ps). Channels are separated by ",". After entering parameters, click "Add Virtual Channel" to add that coincidence strategy to the coincidence calculation. A "coincidence" count is triggered (considered a coincidence event) only when photons appear on all participating channels within the specified window interval; that is, the relationship between channels is "AND".

After starting the measurement, the measurement results for each coincidence strategy are presented in a table format in the right panel. Main indicators include the coincidence count rate, representing counts satisfying the coincidence strategy per unit time (1 second); and the total coincidence count, representing the total count value satisfying that strategy from the start of measurement to the current moment. The chart below plots the historical curve of coincidence count rates for all strategies from the start of measurement to the current moment.

Users can add or delete virtual channels (coincidence strategies). Channels participating in coincidence counting are separated by "," (English comma). To modify an already added virtual channel, double-click the cell that needs modification, make changes, and press Enter.

2.2.12 Selecting Reference Clock

The reference clock is the time-frequency benchmark of the counter. Over longer time scales (greater than 100us), the stability of the reference clock begins to affect measurement results. The counter supports two reference clock settings:

Internal 25 MHz crystal oscillator
External 10 MHz oven-controlled crystal oscillator (OCXO)

The 25 MHz crystal oscillator can be used immediately after power-on and is the counter's default reference clock. If an external clock is needed, ensure the signal connected to the clock interface is a 10M square wave, with a recommended rise time less than 2ns, amplitude less than 3.3V, and jitter less than 10ps. Clocks with poor signal quality or phase noise will degrade the counter's measurement performance.

3 Measurement Procedure

3.1 Cable Connection

1. Power on the host computer.
2. Connect the counter to the host computer using a USB 3.0 male-to-male cable. The counter's USB 3.0 interface is on the front panel. **Pay special attention to connect to a USB 3.0 port on the computer.** If connected to a USB 2.0 port, the system will not work properly.
3. Connect the 10 Gigabit network cable. The supplied cable is an SFP+ DAC 10 Gigabit cable. There are two 10 Gigabit ports on the counter's front panel. The counter's right-side 10 Gigabit port must be connected to the host computer's 10 Gigabit network card port closer to the motherboard. The counter's left-side port must be connected to the host computer's 10 Gigabit network card port farther from the motherboard.
4. Power on the counter.
5. Start the measurement control software.

3.2 Start the Measurement Interface Program

Start the counter host program by double-clicking the desktop shortcut: SIMTURM-TCHPC6

When you need to restart the counter, please first turn off the counter and close the host program. After the counter finishes powering up, then open the host program.

3.3 Set Trigger Threshold Voltage

1. Select the "Parameter Settings" tab.
2. Change the threshold voltage for the corresponding channel, unit millivolts, value is a positive integer, maximum 4096.
3. Click the "Write DAC Values" button.
4. Trigger threshold voltages are not saved after power-off. When the counter restarts, the default threshold voltage for each channel is set to 512 mV.

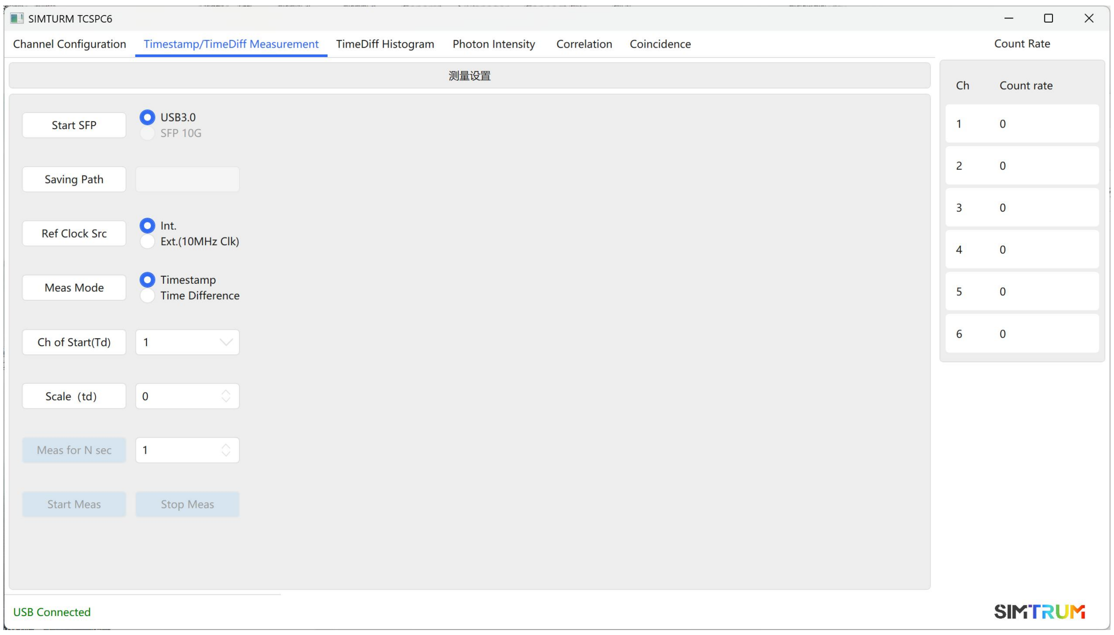
3.4 Channel Delay Adjustment

1. Select the "Parameter Settings" tab.
2. Change the delay compensation value for the corresponding channel. The value can be positive or negative.
3. Click "Update Channel Delays".
4. Channel delay adjustments are not saved after power-off. When the counter restarts, the default delay adjustment for each channel is set to 0 (ps).

3.5 Display Count Rate

1. Click the "Count Rate" tab.
2. Click the "Start Acquisition" button to begin displaying count rates for each channel.
3. Click "Stop" to stop count rate measurement.
4. Please stop count rate measurement before starting other measurement items.
5. When performing count rate measurement, if you click on other tabs, the program will automatically terminate the count rate measurement.

3.6 Timestamp/Time Difference Measurement

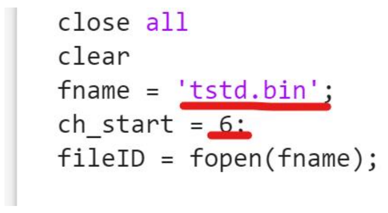


Start SFP: [✓ USB3.0] SFP 10G
Ref Clock Src: [✓ Int.] Ext.(10MHz Clk)
Meas Mode: [✓ Timestamp] Time Difference
Ch of Start(Td): 1
Scale (td): 0
Meas for N sec: 1

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

USB Connected

1. Click the "Timestamp/Time Difference Measurement" tab.
2. Select the output port: USB3.0.
3. Click the "Select Save Path" button to choose the path for saving measurement data files.
4. Select Timestamp mode or Time Difference output.
5. **Gate Open Channel**. The gate open channel in Time Difference mode. Ignored in Timestamp mode.
6. **Scale** magnification factor, only valid in Time Difference mode. Since 64 bits are used to represent time difference with sufficient range, set scale to 0.
7. **Timed Measurement**. In the number box to the right of "Measure N seconds", type the desired measurement duration (unit: seconds). Then click the "Measure N seconds" button. During measurement, this button appears grayed out; it becomes active again when the measurement ends.
8. **Manual Control Measurement**. Click "Start Measurement", and when finished, click the "End Measurement" button.
9. **Parsing Measurement Files**. You can use the provided Matlab scripts to parse the saved binary measurement data files. Use proc_ts_usb.m to parse timestamp data (T2 Timestamp mode). Use proc_t3_usb.m to parse time difference data (T3 mode). Note that you may need to modify the measurement file name and sync channel number in the parsing script:



```
close all
clear
fname = 'tstd.bin';
ch_start = 6;
fileID = fopen(fname);
```

3.7 Real-time Histogram

Refer to the description in section 2.2.9 Real-time Histogram.

3.8 Coincidence Counting

Refer to section 2.2.11 Coincidence Counting.

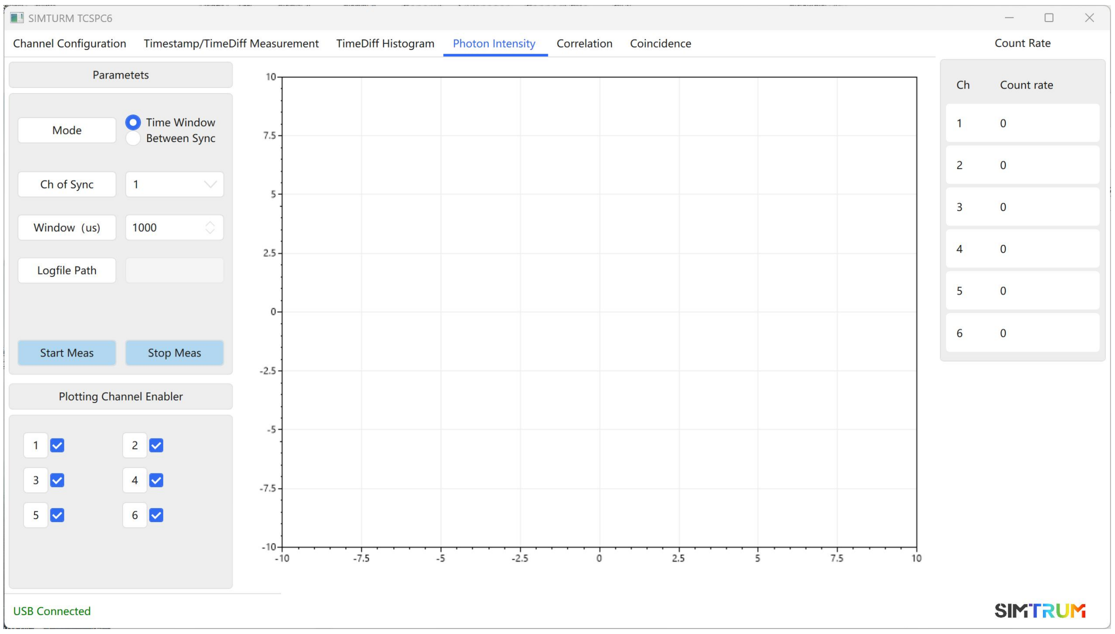
3.9 Photon Counting Statistics

Single photon intensity is similar to count rate measurement, counting the number of photons on a channel over a period of time. Two modes can be selected during measurement.

1. **Time Window Mode**. In this mode, the user needs to specify the statistical window length, unit is us. The counter measures the photon count on each channel within the specified time interval.

2. **Sync Channel Trigger Mode**. In this mode, the user needs to designate the sync signal channel number. The counter measures the number of photons arriving on each channel between two sync signals.

3. After measurement starts, the host software reads the count values output by the counter at a frequency of once per second and plots the curve in real-time. The user can also choose to save the counting results in a binary file for post-measurement analysis.



Mode: [✓ Time Window] Between Sync

Ch of Sync: 1

Window (us): 1000

Logfile Path:

Plotting Channel Enabler: 1 ✓, 2 ✓, 3 ✓, 4 ✓, 5 ✓, 6 ✓

Ch	Count rate
1	0
2	0
3	0
4	0
5	0
6	0

During measurement, first select the statistics mode: Time Window or Sync Signal. Based on the selected mode, choose the sync signal channel number or input the time window length.

If you need to save the measurement results to a file, you need to select the path and filename for the record file.

After setting the measurement parameters, click "Start Measurement". In the plot channel control area, you can enable/disable data plotting for a channel. When a channel is unchecked, its records will not be plotted in the right chart, but this channel's data will still be saved in the record file. Users can change the plotting channel enable/disable status in real-time during measurement.

4 DLL Development Examples

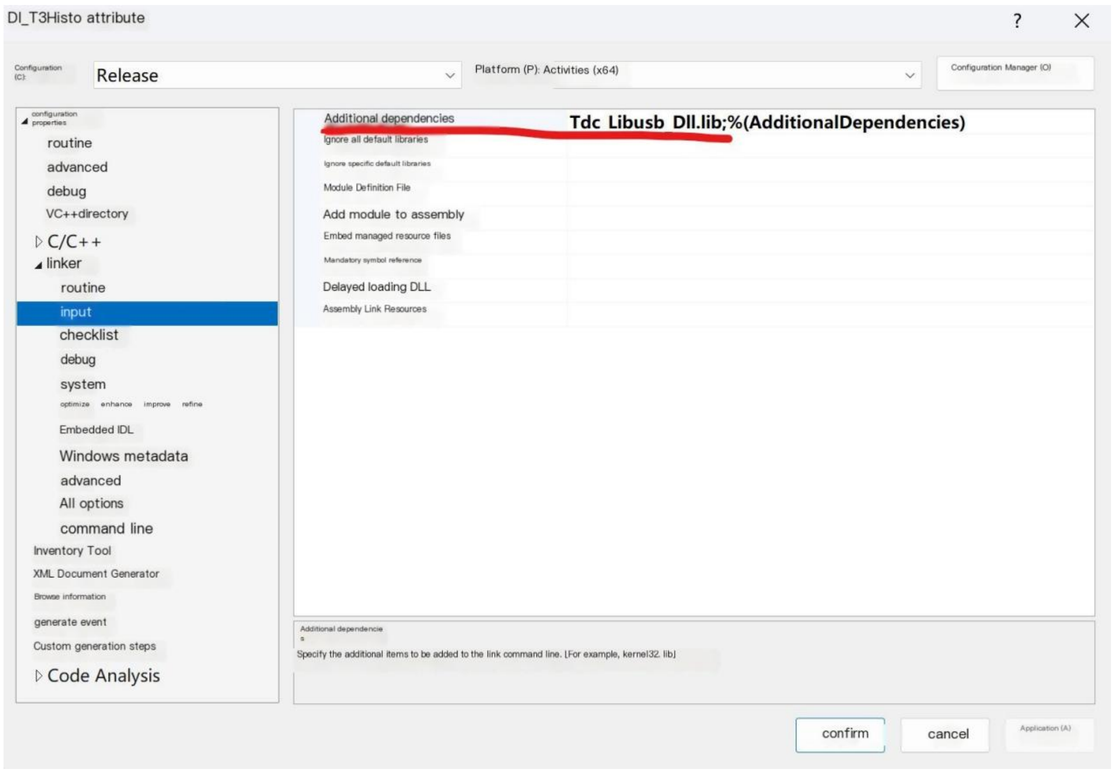
Users can use the provided dynamic link library and header files to develop custom counter control programs on Windows, acquire measurement data online, implement custom automated measurement processes, and data processing applications. Any language supporting DLL calls can be used. Our company provides DLL example programs in languages such as C++/Python/LabVIEW.

The dynamic link library and header files can be found in the installation directory of the host program SIMTURM-TCHPC6. The default installation directory is: C:\Program Files\SIMTURM\SIMTURM-TCHPC6.

Important Note: When calling functions in the DLL, you must first execute LibUsb_Init() to initialize the LibUsb environment. Before exiting the program, you must execute LibUsb_Exit() to exit the USB environment. If LibUsb_Exit() is not executed, an exception will occur the next time LibUsb_Init() is executed. In this case, the only solution is to power cycle the counter.

4.1 Create a New DLL C++ Application

Create a new C++ project in Visual Studio and add the header file Tdc_Libusb_Dll.h. Copy Tdc_Libusb_Dll.dll and Tdc_Libusb_Dll.lib to the project folder. Right-click the project, go to Properties, in the Linker -> Input -> Additional Dependencies, add Tdc_Libusb_Dll.lib, as shown below:



DI_T3Histo attribute

Configuration (C): Release

Platform (P): Activities (x64)

Additional dependencies: Tdc Libusb Dll.lib;%(AdditionalDependencies)

Additional dependencies

Specify the additional items to be added to the link command line. [For example, kernel32. lib]

4.2 Histogram Statistics Example (T3, C++)

This example is written in C++, calling the SDK dynamic link library. It first sets the threshold voltage and channel delays, then sets the histogram calculation parameters and a callback function. In the example, channel 1 is set as the sync channel, and channels 2 and 5 participate in histogram statistics. The counter measures the time difference of channels 2 and 5 relative to channel 1 and performs statistics at specified intervals according to the interface settings (such as bin count and bin width). During example execution, at each specified interval, the program reads the T3 raw data from the host buffer for histogram statistics and executes the callback function. In the callback function, users can read statistical information such as the bin index with the maximum count and the current photon count.

While performing histogram measurement, T3 raw data can also be saved to a file in binary format. There are two ways to save files: one is to pass the complete file path as the first parameter to the `HistoT3Measure` function. Alternatively, as shown in the example, you can call the `getHistoRawdata` function to get T3 data and write it to a file.

The following introduces the code in the example program step by step:

First, set the threshold voltage:

```cpp
dacwrite(i, 500);
```

`i` ranges from 0~5, representing channels 1~6.

Second, set channel delay. Note: here the first parameter range is 1~6, representing channel 1~6.
For example, the following statement compensates the timestamp of channel 1 by subtracting 2000 ps from the original timestamp.

```cpp
setuseroffset(1, -2000);
```

Third, call `HistoT3Measure` to implement histogram measurement. This function has many parameters, prototype as follows:

```cpp
void HistoT3Measure(const char filename[], int mode, int chnum, int sync, int* hist_channels, int h
ist_chcount, int scale, int binsize, int binnum, int mt_ms, int period_ms, CountReady callback = n
ullptr);
```

`const char filename[]`: Target path to save T3 file. Avoid Chinese characters in the path.
`int mode`: Histogram mode, 0 for accumulate mode, 1 for refresh mode.
`int chnum`: Maximum number of channels for the counter, can be directly written as 10.
`int sync`: Sync channel.
`int* hist_channels`: Array of channels participating in histogram statistics.
`int hist_chcount`: Number of channels participating in histogram statistics.
`int scale`: Scaling factor for time difference. No scaling needed, set to 0.
`int binsize`: Histogram bin size, unit ps.
`int binnum`: Number of histogram bins.
`int mt_ms`: Measurement duration, unit milliseconds.
`int period_ms`: Calculation interval, unit milliseconds. Histogram statistics are performed every specified interval.

CountReady callback: Callback function, called after each histogram calculation interval.

The example call parameters are set as follows:

cpp

```cpp
HistoT3Measure("", 0, 10, 1, channels, 2, scale, binsize, binnum, 3100, 500, print_histmax);
```

Save file path is empty string (no file saved); Count mode: accumulate; Maximum channels 10; Sync channel 1; Participating channels: int channels[2] = { 2,5 }; Number of participating channels: 2; Scaling factor 0; Measurement duration 3100 ms; Statistics interval 500 ms; Callback function: print_histmax.

In the callback function, read the bin index with the maximum count for channels 2 and 5, the count value, and the number of bytes read in this cycle, and print them.

int getHistoMax(int chan): chan takes values 1~6, returns the index of the bin with the maximum count for that channel.
UINT32 getHistoHitCountAtCh(int ch): ch takes values 1~6, returns the count value for the specified channel in this cycle.
int getHistoProcBytes(): Returns the number of bytes of data processed in this calculation cycle.

Fourth, stop measurement:

cpp

```cpp
stopHistoT3Measure();
```

Fifth: Get all count values for channel 2 histogram and print:

cpp

```cpp
getBinCount(2, buffer, binnum);
```

Where buffer is a pointer to a buffer whose size is at least binnum*4.

The complete code is as follows:

cpp

```cpp
#include <iostream>#include <vector>#include <thread>#include <iostream>#include <fstream>#inclu
de <sstream>#include "Tdc_Libusb_Dll.h"

#define BUF_SIZE (1024*1024*128)char destBuffer[BUF_SIZE];volatile bool stop = false;using nam
espace std;


UINT32 maxhistory[2][1024];int byteshistory[1024];int cntshistory[2][1024];int index = 0;

// Callback function, passed as parameter to HistoT3Measure. It is called after each histogram ca
lculation.void print_histmax(){

    // Get the index of the bin with the maximum count for channels 2/5
    maxhistory[0][index] = getHistoMax(2);

    maxhistory[1][index] = getHistoMax(5);

    // Get the byte count of raw data

    byteshistory[index] = getHistoProcBytes();

    // Get the photon count for channels 2/5

    cntshistory[0][index] = getHistoHitCountAtCh(2);

    cntshistory[1][index] = getHistoHitCountAtCh(5);

    printf("max is %d, %d\r\n", maxhistory[0][index], maxhistory[1][index]);

    index++;

    int bytes_retrieved;

    // Get raw T3 data, copy to destBuffer, for user-defined processing.

    // The histogram measurement function itself can save T3 raw data to a file,

    // so the function below is not needed unless real-time raw data processing is required.

    getHistoRawdata(destBuffer, &bytes_retrieved);

    printf("bytes retrieved is %d, \r\n", bytes_retrieved);}

int main(){

    std::cout << "Hello HistoT3 Test!\n";

    int ret = -1;

    // Initialize USB environment

    LibUsb_Init();

    // Set threshold voltage 500mV, dacwrite(0,xxx) sets threshold for channel 1

    for (int i = 0; i < 6; i++)

    {

        ret = dacwrite(i, 500);

        if (ret != 0)

        {

            printf("write threshold voltage for channel %d failed\r\n", i+1);

            return -1;

        }

    }
    ret = 0;

    // Set channel delays

    ret += setuseroffset(1, -2000);

    ret += setuseroffset(2, 0);

    ret += setuseroffset(3, 0);

    ret += setuseroffset(4, 0);

    ret += setuseroffset(5, 0);

    ret += setuseroffset(6, 0);

    if (ret != 0)

    {

        printf("write channel offset failed\r\n");

        return -1;

    }

    // Start measurement

    // Define channels participating in histogram statistics, here channels 2 and 5

    int channels[2] = { 2,5 };

    // Number of histogram bins

    int binnum = 1000;

    // bin size, unit ps

    int binsize = 10;

    int scale = 0;

    index = 0;

    // Start histogram measurement

    // 1st param filename, path to save T3 raw data. When empty, T3 raw data is not saved.

    // 2nd param mode, 0 for accumulate mode, 1 for refresh mode.

    // 3rd param chnum, total number of channels, can use fixed value 10.

    // 4th param sync, sync channel number, 1 means channel 1 is sync channel.

    // 5th param hist_channels, array of channels participating in histogram statistics.

    // 6th param hist_chcount, number of channels participating in histogram stats. Can be the si
ze of the array, or smaller. E.g., array length 3, this param can be 1,2, or 3.
```

```
    // 7th param scale, scaling factor. Since 64-bit timestamp has sufficient range, scaling can be
 fixed at 0.

    // 8th param binsize

    // 9th param binnum, number of bins. binsize * binnum is the maximum range of histogram
stats. Can be modified as needed. More bins increase computational load.

    // 10th param mt_ms, measurement duration, unit milliseconds (ms).

    // 11th param period_ms, histogram statistics time interval, unit milliseconds (ms). Specifies h
ow often to read raw data and perform histogram stats.

    //     At high count rates, consider that data volume should not exceed internal buffer (256
MB).

    //     E.g., count rate 32M, data per second 128MB, so recommended interval <2 sec. 500
ms~1000ms interval is recommended.

    // 12th param callback function. This parameter can be omitted, meaning no callback needed.
 Called after each histogram calculation.

    HistoT3Measure("", 0, 10, 1, channels, 2, scale, binsize, binnum, 3100, 500, print_histmax);

    std::this_thread::sleep_for(std::chrono::milliseconds(3200));

    // Stop histogram measurement

    stopHistoT3Measure();

    // Prepare buffer to store histogram data

    uint32_t* buffer = (uint32_t*)malloc(4 * binnum);

    // Get histogram statistics for channel 2

    getBinCount(2, buffer, binnum);

    // Print histogram data

    for (int i = 0; i < binnum; i++)

    {

        printf("%d, %d %u\r\n", i, (i + 1) * binsize, buffer[i]);

    }

    // Print some historical information recorded in the callback

    for (int i = 0; i < index; i++)

    {

        printf("ch2: max %u, counts %d, bytes %d,\r\n", maxhistory[0][i], cntshistory[0][i], byteshist
ory[i]);
```

```
        printf("ch5: max %u, counts %d, bytes %d,\r\n", maxhistory[1][i], cntshistory[1][i], byteshist
ory[i]);

    }

    printf("Done! Exit\r\n");

    /* // Example for channel 3

    getBinCount(3, buffer, binnum);

    for (int i = 0; i < 1000; i++)

    {

        printf("%d, %d %u\r\n", i, (i + 1) * binsize, buffer[i]);

    }

    */

    printf("Done! Exit\r\n");

    // Exit USB environment

    LibUsb_Exit();}
```

4.3 Histogram Statistics Example (T2, C++)

This example has the same functionality as the T3 mode histogram statistics example in the previous chapter. The difference is that the T3 mode histogram is based on T3 data output by the counter, while this example is based on T2 (timestamp) data output by the counter. In some LIDAR applications, one laser pulse may generate a large number of photons on a single echo channel. In this case, the T3 mode designed for single-photon applications may not correctly handle the large number of photons within one sync signal cycle. In such cases, it is recommended to use T2 mode histogram based on timestamps.

In T2 mode histogram, the time difference relative to the sync signal and histogram statistics are calculated based on timestamps, without requiring the single-photon assumption, making it more general-purpose.

When using T2 histogram statistics, only two functions differ from T3 mode: HistoT2Measure to start measurement and stopHistoT2Measure to stop measurement. The parameters of HistoT2Measure are consistent with its T3 version and will not be detailed here; please refer to the previous chapter.

The T2 histogram statistics function, like the T3 version, can also specify a file path to save raw data. In each statistics cycle, raw data can also be acquired. The difference from T3 is that the raw data here is not T3 format time difference data, but T2 format timestamp data.

The source code is provided below for reference:

cpp

```cpp
#include <iostream>#include <vector>#include <thread>#include <iostream>#include <fstream>#inclu
de <sstream>#include "Tdc_Libusb_Dll.h"

#define BUF_SIZE (1024*1024*128)char destBuffer[BUF_SIZE];volatile bool stop = false;using nam
espace std;


UINT32 maxhistory[2][1024];int byteshistory[1024];int cntshistory[2][1024];int index = 0;

// Callback function, passed as parameter to HistoT2Measure. It is called after each histogram ca
lculation.void print_histmax(){

    // Get the index of the bin with the maximum count for channels 2/3

    maxhistory[0][index] = getHistoMax(2);

    maxhistory[1][index] = getHistoMax(3);

    // Get the byte count of raw data

    byteshistory[index] = getHistoProcBytes();

    // Get the photon count for channels 2/3

    cntshistory[0][index] = getHistoHitCountAtCh(2);

    cntshistory[1][index] = getHistoHitCountAtCh(3);

    printf("max is %d, %d\r\n", maxhistory[0][index], maxhistory[1][index]);

    index++;

    int bytes_retrieved;

    // Get raw T2 data, copy to destBuffer, for user-defined processing.

    // The T2 histogram measurement function itself can save T2 raw data to a file,

    // so the function below is not needed unless real-time raw data processing is required.

    getHistoRawdata(destBuffer, &bytes_retrieved);

    printf("bytes retrieved is %d, \r\n", bytes_retrieved);}

int main(){

    std::cout << "Hello HistoT2 Test!\n";

    int ret = -1;

    // Initialize USB environment

    LibUsb_Init();

    // Set threshold voltage 500mV, dacwrite(0,xxx) sets threshold for channel 1
```

```
for (int i = 0; i < 6; i++)
{
    ret = dacwrite(i, 500);
    if (ret != 0)
    {
        printf("write threshold voltage for channel %d failed\r\n", i+1);
        return -1;
    }
}
ret = 0;
// Set channel delays
ret += setuseroffset(1, -3000);
ret += setuseroffset(2, 0);
ret += setuseroffset(3, 0);
ret += setuseroffset(4, 0);
ret += setuseroffset(5, 0);
ret += setuseroffset(6, 0);
if (ret != 0)
{
    printf("write channel offset failed\r\n");
    return -1;
}
// Start measurement
// Define channels participating in histogram statistics, here channels 2 and 3
int channels[2] = { 2,3 };
// Number of histogram bins
int binnum = 100000;
// bin size, unit ps
int binsize = 10;
int scale = 0;
```

```
    index = 0;

    // Start histogram measurement

    // 1st param filename, path to save T2 raw data. When empty, T2 raw data is not saved.

    // 2nd param mode, 0 for accumulate mode, 1 for refresh mode.

    // 3rd param chnum, total number of channels, can use fixed value 10.

    // 4th param sync, sync channel number, 1 means channel 1 is sync channel.

    // 5th param hist_channels, array of channels participating in histogram statistics.

    // 6th param hist_chcount, number of channels participating in histogram stats.

    // 7th param scale, scaling factor. Since 64-bit timestamp has sufficient range, scaling can be
 fixed at 0.

    // 8th param binsize

    // 9th param binnum, number of bins. binsize * binnum is the maximum range of histogram
stats.

    // 10th param mt_ms, measurement duration, unit milliseconds (ms).

    // 11th param period_ms, histogram statistics time interval, unit milliseconds (ms).

    // 12th param callback function.

    //HistoT2Measure("", 0, 10, 1, channels, 2, scale, binsize, binnum, 3100, 500, print_histmax);//
no dump to file

    HistoT2Measure("D:\\test_data\\t2hist.bin", 0, 10, 1, channels, 2, scale, binsize, binnum, 3100,
500, print_histmax);//dump to file

    std::this_thread::sleep_for(std::chrono::milliseconds(3200));

    // Stop histogram measurement

    stopHistoT2Measure();

    // Prepare buffer to store histogram data

    uint32_t* buffer = (uint32_t*)malloc(4 * binnum);

    // Get histogram statistics for channel 3

    getBinCount(3, buffer, binnum);

    // Print histogram data

    for (int i = 0; i < 1000; i++)

    {

        printf("%d, %d %u\r\n", i, (i + 1) * binsize, buffer[i]);
    }
    // Print some historical information recorded in the callback
    for (int i = 0; i < index; i++)
    {
        printf("ch2: max %u, counts %d, bytes %d,\r\n", maxhistory[0][i], cntshistory[0][i], byteshist
ory[i]);
        //printf("ch5: max %u, counts %d, bytes %d,\r\n", maxhistory[1][i], cntshistory[1][i], byteshis
tory[i]);
    }
    printf("Done! Exit\r\n");
    // Exit USB environment
    LibUsb_Exit();}
```

4.4 Histogram Mapping Example (C++)

This example demonstrates how to perform histogram measurements multiple times in a loop, simulating a test scenario requiring mapping/scanning. Each measurement acquires histogram statistical information and saves the measurement data in a separate file. The example performs 20 loop measurements, with raw data saved in 20 different binary files.

In the example, channel 1 is the sync channel, and channels 2 and 5 participate in histogram statistics. Channel 2 statistics are read and printed after each measurement is completed. Channel 5 measurement data is read and recorded in the callback function and printed after all 20 measurements are completed. The complete source code is as follows:

cpp

```
#include <iostream>#include <vector>#include <thread>#include <iostream>#include <fstream>#inclu
de <sstream>#include "Tdc_Libusb_Dll.h"

#define BUF_SIZE (1024*1024*128)char destBuffer[BUF_SIZE];//volatile bool stop = false;using na
mespace std;


UINT32 maxhistory[1024];int byteshistory[1024];int cntshistory[1024];int index = 0;

std::ofstream outF;uint64_t totalbytes = 0;

// Callback function, passed as parameter to HistoT3Measure. It is called after each histogram ca
lculation.void print_histmax();

int main(){
```

```
std::cout << "Hello HistoT3 Mapping Test!\n";

int ret = -1;

// Initialize USB environment

LibUsb_Init();

// Set threshold voltage 500mV, dacwrite(0,xxx) sets threshold for channel 1

for (int i = 0; i < 6; i++)

{

    ret = dacwrite(i, 500);

    if (ret != 0)

    {

        printf("write threshold voltage for channel %d failed\r\n", i + 1);

        return -1;

    }

}

ret = 0;

// Set channel delays

ret += setuseroffset(1, -2000);

ret += setuseroffset(2, 0);

ret += setuseroffset(3, 0);

ret += setuseroffset(4, 0);

ret += setuseroffset(5, 0);

ret += setuseroffset(6, 0);

if (ret != 0)

{

    printf("write channel offset failed\r\n");

    return -1;

}

// Exit USB environment for this example's structure (Note: Might need re-init inside loop)

LibUsb_Exit();
```

```
// Define channels participating in histogram statistics, here channels 2 and 5

int channels[2] = { 2,5 };

// Number of histogram bins

int binnum = 1000;

// bin size, unit ps

int binsize = 10;

int scale = 0;

index = 0;

totalbytes = 0;

char filepath[80] = "D:\\temp\\t3loop\\";

for (int i = 0; i < 20; i++)

{

    char filename[20];

    char fullpath[80];

    strcpy(fullpath, filepath);

    sprintf(filename, "t3data_%d.bin", i + 1);

    strcat(fullpath, filename);

    printf("this is %d turn\r\n", i);

    // Initialize USB environment for each loop iteration

    LibUsb_Init();

    // Start histogram measurement, saving raw T3 data to file specified by fullpath

    HistoT3Measure(fullpath, 0, 10, 1, channels, 2, scale, binsize, binnum, 100000, 500, print
_histmax);

    std::this_thread::sleep_for(std::chrono::milliseconds(1100));

    // Stop histogram measurement

    stopHistoT3Measure();

    // Get the index of the bin with the maximum count for channel 2 and print

    UINT32 max = getHistoMax(2);

    printf("ch2: max index is %u\r\n", max);

    // Exit USB environment for this loop iteration
```

```
        LibUsb_Exit();
    }
    // Print some historical information recorded in the callback function
    for (int i = 0; i < index; i++)
    {
        printf("ch5: max %u, bytes %d, counts %d\r\n", maxhistory[i], byteshistory[i], cntshistory[i]);
    }
    printf("Measurement Done! Exit!\r\n");}
// Callback function, passed as parameter to HistoT3Measure. It is called after each histogram ca
lculation.void print_histmax(){
    // Get the index of the bin with the maximum count for channel 5
    maxhistory[index] = getHistoMax(5);
    // Get the byte count of raw data
    byteshistory[index] = getHistoProcBytes();
    // Get the photon count for channel 5
    cntshistory[index] = getHistoHitCountAtCh(5);
    index++;
    // Optional: get raw T3 data for real-time processing (commented out as saving is handled b
y HistoT3Measure)
    // int bytes_retrieved;
    // getHistoRawdata((void*)destBuffer, &bytes_retrieved);
    // outF.write(destBuffer, bytes_retrieved);}
```

4.5 T2 Histogram Scan Example (C++)

This example is written in C++ under Visual Studio 2022. It demonstrates how to combine scanning sync signals (pixel, line, frame clocks) for histogram scanning, performing histogram statistics for each pixel and attaching pixel, line, frame information to the results.

The example uses the `HistoT2ScanMeasure_usb` function to start measurement and `stopHistoT2ScanMeasure_usb` to stop measurement.

During measurement, by default, channels 4, 5, and 6 are for pixel sync, line sync, and frame sync signals (pulses) respectively. The reference light is on channel 1, and the photon channel is channel 2. The data update interval is 0.5 seconds. Measurement starts and stops after 3.5 seconds. During measurement, histogram statistical results for each pixel are acquired every 0.5 seconds and saved in binary format in the file `d:\\test_data\\histdata.bin`.

The histogram data format for each pixel consists of a header and histogram data block. The first 32 bytes are the header, composed of 4 int64 numbers representing:

`header[0]` = Timestamp of the pixel sync signal.
`header[1]` = Pixel count.
`header[2]` = Line count.
`header[3]` = Frame count.

Following the header is the histogram data block. The size of the data block is determined by the number of bins, counting depth (in this software version, counting depth is 32 bits, 4 bytes), and the number of channels participating in histogram statistics. For example, in this example, the number of bins is 1000, counting depth is 4, and number of participating channels is 1, so the histogram data block size is 1000 * 4 = 4000 bytes. These 4000 bytes represent 1000 uint32 values, which is the histogram statistical result for that channel at that specific pixel.

This example also provides a Matlab script `proc_histscanlog.m` to parse the saved pixel histogram data. Note that two parameters in the script—the number of bins and the number of counter channels—must match the parameters used during measurement:

matlab

```
binnum = 10000;

chn_cnt = 1;
```

The C++ program source code is provided below for reference:

cpp

```
#include <iostream>#include <vector>#include <thread>#include <fstream>#include <atomic>#includ
e <sstream>#include "Tdc_Libusb_Dll.h"using namespace std;

#define BUF_SIZE (1024*1024*128)char destBuffer[BUF_SIZE];

UINT32 maxhistory[2][1024];int cntshistory[2][1024];int byteshistory[1024];

atomic<int> call_cnt(0);

atomic<bool> stop(false);//volatile bool stop = false;

std::string filename;

std::ofstream outFile;uint64_t total_bytes = 0;int index = 0;

void t2scan_callback(){

    call_cnt.fetch_add(1);}
```

```
void w2file(){
    while (true)
    {
        int cnt = call_cnt.load();
        if (cnt > 0)
        {
            int rbytes = get_histo_mmf_data(BUF_SIZE, destBuffer);
            outFile.write(destBuffer, rbytes);
            total_bytes = total_bytes + rbytes;
            call_cnt.fetch_add(-1);
        }
        else if (stop.load())
        {
            break;
        }
    }}
int main(){
    std::cout << "Hello T2ScanHisto Test!\n";
    int ret = -1;
    // Initialize USB environment
    LibUsb_Init();
    // Set threshold voltage (example shows specific settings for some channels)
    dacwrite(0, 300); // Ch1
    dacwrite(1, 300); // Ch2
    dacwrite(3, 800); // Ch4 (Note: index 3 is channel 4)
    ret = 0;
    // Set channel delays
    ret += setuseroffset(1, -3000);
    ret += setuseroffset(2, 0);
    ret += setuseroffset(3, 0);
    ret += setuseroffset(4, 0);

    ret += setuseroffset(5, 0);

    ret += setuseroffset(6, 0);

    if (ret != 0)

    {

        printf("write channel offset failed\r\n");

        return -1;

    }

    // Start measurement

    filename.assign("d:\\test_data\\histdata.bin");

    outFile.open(filename, std::ios::binary | std::ios::trunc);

    // Define channels participating in histogram statistics, here channel 2 (but array defined with
{2,2})

    int channels[2] = { 2,2 };

    // Number of histogram bins

    int binnum = 10000;

    // bin size, unit ps

    int binsize = 10;

    index = 0;

    call_cnt.store(0);

    stop.store(false);

    std::thread wthread(w2file);

    // Start histogram measurement

    /// <summary>

    /// using T2 and pixel, line, frame sync pulse to calculate histogram for each pixel

    /// </summary>

    /// <param name="filename">file path to record raw T2(timestamp) binary data, if ="", then no
data will be recorded</param>

    /// <param name="chnum">how many channels the single photon counter has</param>

    /// <param name="sync">the channel number of the reference signal</param>
    /// <param name="hist_channels"> array of int, the channels involved in the histogram calcula
tion</param>

    /// <param name="hist_chcount">how many channels are involved in the histogram calculation
</param>

    /// <param name="scale">always 0</param>

    /// <param name="binsize">bin size</param>

    /// <param name="binnum"> how many bins are there in the histogram</param>

    /// <param name="mt_ms">duration of the measurement</param>

    /// <param name="period_ms">interval of the data update</param>

    /// <param name="callback">callback when data is updated</param>

    /// <returns></returns>

    HistoT2ScanMeasure_usb("", 6, 1, channels, 1, 0, binsize, binnum, 6500, 500, t2scan_callbac
k);//no T2 recording, 2channel, sync ==1, stop ==2, scale = 0 binsize, binnumber, 6.5s measure
ment duration, 0.5 data update interval, callback

    Sleep(3500);

    stopHistoT2ScanMeasure_usb();

    stop.store(true);

    wthread.join();

    outFile.flush();

    outFile.close();

    printf("log histdata %llu\r\n", total_bytes);

    printf("Done! Exit\r\n");

    // Exit USB environment

    LibUsb_Exit();}
```

4.6 T2 Photon Count Scan Example (C++)

This example is written in C++ under Visual Studio 2022. It demonstrates how to combine scanning sync signals (pixel, line, frame clocks) to perform photon count scanning for each channel (excluding the pixel, line, frame clock channels), counting photons in each pixel (between two pixel clock pulses) and attaching pixel, line, frame information to the results.

The example uses the `T2CScanMeasure_usb` function to start measurement and `stopT2CScanMeasure_usb` to stop measurement.

When performing photon counting statistics in scan mode, by default, channels 4, 5, and 6 are for pixel sync, line sync, and frame sync signals (pulses) respectively. In this example, the data update interval is 0.5 seconds. Measurement starts and stops after 3.5 seconds. During measurement, photon count (light intensity) results for each pixel are acquired every 0.5 seconds and saved in binary format in the file `d:\\test_data\\countdata.bin`.

The histogram data format for each pixel consists of a header and a data block. The first 32 bytes are the header, composed of 4 int64 numbers representing:

`header[0]` = Timestamp of the pixel sync signal.
`header[1]` = Pixel count.
`header[2]` = Line count.
`header[3]` = Frame count.

Following the header is the photon count data block. Each channel of the counter has an independent counter to count photons. The counter bit width is 64 bits, 8 bytes. Therefore, the data block size is number of channels * 8 bytes. The second parameter of the `T2CScanMeasure_usb` function is the number of channels. In this example, the number of channels is 6, so the data block size is 6 * 8 = 48 bytes.

This example also provides a Matlab script `proc_countscanlog.m` to parse the saved photon count statistics data. Note that the parameter `chn_cnt` (number of channels) in the script must match the parameter used during measurement:

matlab

```
chn_cnt = 6;
```

The C++ program source code for the example is provided below for reference:

cpp

```
#include <iostream>#include <vector>#include <thread>#include <fstream>#include <atomic>#includ
e <sstream>#include "Tdc_Libusb_Dll.h"using namespace std;

#define BUF_SIZE (1024*1024*16)char destBuffer[BUF_SIZE];

UINT32 maxhistory[2][1024];int cntshistory[2][1024];int byteshistory[1024];

atomic<int> call_cnt(0);

atomic<bool> stop(false);//volatile bool stop = false;

std::string filename;

std::ofstream outFile;uint64_t total_bytes = 0;int index = 0;

void t2scan_callback(){

    call_cnt.fetch_add(1);}

void w2file(){

    while (true)
```

```
        {
            int cnt = call_cnt.load();
            if (cnt > 0)
            {
                int rbytes = get_histo_mmf_data(BUF_SIZE, destBuffer);
                outFile.write(destBuffer, rbytes);
                total_bytes = total_bytes + rbytes;
                call_cnt.fetch_add(-1);
            }
            else if (stop.load())
            {
                break;
            }
        }}
    int main(){
        std::cout << "Hello T2ScanCount Test!\n";
        int ret = -1;
        // Initialize USB environment
        LibUsb_Init();
        // Set threshold voltage 500mV for all channels
        for (int i = 0; i < 6; i++)
        {
            ret = dacwrite(i, 500);
            if (ret != 0)
            {
                printf("write threshold voltage for channel %d failed\r\n", i + 1);
                return -1;
            }
        }
        ret = 0;
        // Set channel delays

        ret += setuseroffset(1, 0);

        ret += setuseroffset(2, 0);

        ret += setuseroffset(3, 0);

        ret += setuseroffset(4, 0);

        ret += setuseroffset(5, 0);

        ret += setuseroffset(6, 0);

        if (ret != 0)

        {

            printf("write channel offset failed\r\n");

            return -1;

        }

        // Start measurement

        filename.assign("d:\\test_data\\countdata.bin");

        outFile.open(filename, std::ios::binary | std::ios::trunc);

        index = 0;

        call_cnt.store(0);

        stop.store(false);

        std::thread wthread(w2file);

        // Start measurement

        /// <summary>

        /// using T2 and pixel, line, frame sync pulse to calculate photon count for each pixel

        /// </summary>

        /// <param name="filename">file path to record raw T2(timestamp) binary data, if ="", then no
    data will be recorded</param>

        /// <param name="chnum">how many channels the single photon counter has</param>

        /// <param name="mt_ms">duration of the measurement</param>

        /// <param name="period_ms">interval of the data update</param>

        /// <param name="callback">callback when data is updated</param>

        /// <returns></returns>
        T2CScanMeasure_usb("", 6, 6500, 500, t2scan_callback);//no T2 recording, 2channel, sync ==
    1, stop ==2, scale = 0 binsize, binnumber, 6.5s measurement duration, 0.5 data update interval,
    callback

        Sleep(3500);

        stopT2CScanMeasure_usb();

        stop.store(true);

        wthread.join();

        outFile.flush();

        outFile.close();

        printf("log countdata %llu\r\n", total_bytes);

        printf("Done! Exit\r\n");

        // Exit USB environment

        LibUsb_Exit();}
```

4.7 Coincidence Counting Example (C++)

This example is written in C++ under Visual Studio 2022. It demonstrates how to set channel trigger threshold voltages, channel delays, and create two virtual channels, i.e., coincidence counting strategies. Virtual channel 1 is the coincidence of channels 1 and 2, with a coincidence window of 1975 ps. The second virtual channel is the coincidence of channels 1 and 5, with a coincidence window of 1990 ps. The coincidence measurement duration is 3100 ms, reading data every 500 ms, and counting coincidence counts and calculating coincidence count rates. In the callback function, coincidence count values, coincidence count rates, coincidence calculation execution time, and other data are acquired, printed, saved, and historical data is printed at the end of the measurement.

The function to start coincidence counting measurement is:

cpp

```
void startCoinMeasure(int (*chan_list)[7], UINT64* windows_size, int list_row, int mt_ms, int update
_rate, CountReady callback = nullptr);
```

1st parameter `chan_list`: A 2D array with 7 columns, representing the composition of virtual channels, i.e., coincidence strategies.
2nd parameter `windows_size`: Array of window sizes, specifying the coincidence window size for each strategy.
3rd parameter `list_row`: Number of virtual channels.
4th parameter `mt_ms`: Measurement duration (milliseconds).
5th parameter `update_rate`: Coincidence calculation interval time (milliseconds).

6th parameter callback: Callback function, called after each coincidence calculation. This parameter can be omitted.

In the example program, the first parameter is a 2-row by 7-column 2D array. Each row represents one virtual channel, i.e., one coincidence strategy. The first element of each row indicates how many channels constitute this virtual channel. In this example, it is 2, meaning coincidence between 2 channels. The next 6 elements of each row indicate the channels participating in the coincidence calculation. For the first virtual channel, the 2nd and 3rd elements of this row are 1 and 2, indicating channels 1 and 2 constitute this virtual channel.

cpp

```
// Set 2 virtual channels, first: 1 and 2, second: 1 and 5int chan_list[2][7]; // First dimension 2 means two virtual channels. If there are 3 virtual channels, this would be 3. 7 is fixed.

chan_list[0][0] = 2; // Indicates two channels participate in coincidence

chan_list[0][1] = 1;

chan_list[0][2] = 2;

chan_list[1][0] = 2; // Indicates two channels participate in coincidence

chan_list[1][1] = 1;

chan_list[1][2] = 5;
```

The second parameter is an array representing the coincidence window size for the corresponding strategy. In the example, this array is:

cpp

```
UINT64 window_list[2] = { 1975,1990 };
```

This indicates the coincidence window for the first strategy (between 1,2) is 1975 ps, and for the second strategy (between 1,5) is 1990 ps. The length of this array should equal the number of strategies, i.e., the row count of the first parameter.

In the example, a callback function void print_coinRate() is defined and passed as the last parameter to startCoinMeasure. It is called after each coincidence calculation. In this function, getCoinCount(0) is called to get the count value for the first strategy (getCoinCount(1) returns the count for the second strategy), and getCoinRate(0) to get the count rate for the first strategy. The acquired count values are saved in an array of sufficient length. After measurement ends, the coincidence count values and rates throughout the test process are printed.

The source code is as follows:

cpp

```
#include <iostream>#include "Tdc_Libusb_Dll.h"#include <vector>#include <thread>using namespace std;
```

```
volatile uint64_t countHistory[10][1024];volatile double rateHistory[10][1024];volatile uint64_t etimeHi
story[1024];volatile uint32_t exetimeHistory[1024];volatile int rbytesHistory[1024];volatile int call_inde
x;

ULONGLONG time_start, time_end;

void print_coinRate();

int main(){

    std::cout << "Hello DLL Coincidence Test!\n";

    vector<uint64_t> CoinCnt;

    int ret = 0;

    // Initialize USB environment

    LibUsb_Init();

    printf("set trigger voltage\r\n");

    // Set threshold voltage 500mV. Note: the first parameter 0 represents channel 1.

    for (int i = 0; i < 6; i++)

    {

        ret = dacwrite(i, 500);

        if (ret != 0)

        {

            printf("set threshold voltage for channel %d failed\r\n", i + 1);

        }

    }

    printf("set channel delay\r\n");

    // Set channel delays

    ret += setuseroffset(1, -2000);

    ret += setuseroffset(2, 0);

    ret += setuseroffset(3, 0);

    ret += setuseroffset(4, 0);

    ret += setuseroffset(5, 0);

    ret += setuseroffset(6, 0);

    if (ret != 0)
    {
        printf("set channel offset failed!\r\n");
    }
    // Set 2 virtual channels, first: 1 and 2, second: 1 and 5
    int chan_list[2][7]; // First dimension 2 means two virtual channels. 7 is fixed.
    chan_list[0][0] = 2; // Indicates two channels participate in coincidence
    chan_list[0][1] = 1;
    chan_list[0][2] = 2;
    chan_list[1][0] = 2; // Indicates two channels participate in coincidence
    chan_list[1][1] = 1;
    chan_list[1][2] = 5;
    // Use different coincidence windows for the two virtual channels
    UINT64 window_list[2] = { 1975,1990 };
    call_index = 0;
    printf("start measure\r\n");
    // Start coincidence measurement
    // 1st param chan_list, virtual channel array.
    // 2nd param windows_size, window size array.
    // 3rd param list_row, number of virtual channels.
    // 4th param mt_ms, measurement duration (ms).
    // 5th param update_rate, coincidence calculation interval (ms).
    // 6th param callback, callback function called after each coincidence calculation. Can be omi
tted.
    startCoinMeasure(chan_list, window_list, 2, 3100, 500, print_coinRate);
    time_start = GetTickCount64();
    std::this_thread::sleep_for(std::chrono::milliseconds(3200));
    // Stop coincidence measurement
    stopCoinMeasure();
    for (int i = 0; i < call_index; i++)
    {
        printf("elapsed time: %llu , exe time %u, read bytes: %d\r\n", etimeHistory[i], exetimeHisto
ry[i], rbytesHistory[i]);
        printf("%d, coinRate0: %3.2e，  coinRate1: %3.2e\r\n", i, rateHistory[0][i], rateHistory[1][i]);
        printf("%d, coinCount0 %llu，  coinCount %llu\r\n", i, countHistory[0][i], countHistory[1][i]);
    }
    // Exit USB environment
    LibUsb_Exit();}
void print_coinRate(){
    time_end = GetTickCount64();
    // Read count value for the 1st virtual channel (strategy). Needs to be read each cycle.
    // Reading resets the channel's count value; otherwise, counts accumulate.
    countHistory[0][call_index] = getCoinCount(0);
    countHistory[1][call_index] = getCoinCount(1);
    // Get coincidence count rate for the 1st virtual channel (strategy)
    double cnt0 = getCoinRate(0);
    double cnt1 = getCoinRate(1);
    rateHistory[0][call_index] = cnt0;
    rateHistory[1][call_index] = cnt1;
    // Get number of bytes processed
    rbytesHistory[call_index] = getCoinProcBytes();
    // Milliseconds elapsed since start of measurement
    etimeHistory[call_index] = time_end - time_start;
    // Coincidence calculation execution time, unit ms
    exetimeHistory[call_index] = getCoinElapsedTime();
    call_index++;
    printf("time elapsed %llu\r\n", (time_end - time_start));}
```

4.8 Histogram Statistics (Python)

This example uses the ctypes library in a Windows Python environment to import the dynamic link library.

First, set the trigger threshold and apply a -2000 ps delay adjustment to channel 1 (to facilitate observation of histogram data). The example uses the T3 histogram statistics function HistoT3Measure, setting channel 1 as the sync channel, and channels 2 and 3 to participate in histogram statistics, analyzing the distribution of their time differences relative to channel 1.

Set bin size to 10 ps, number of bins to 1000, measurement duration to 3100 ms, and read buffer data for histogram statistics every 500 ms. Pass the callback function c_callback, which is called after each histogram calculation, printing the index of the bin with the maximum count for channels 2 and 3.

After measurement ends, get all bin count values for channel 2 and print them.

The source code is provided below for reference:

python

```
# coding=gb2312from ast import Callimport ctypesfrom pickle import NONEimport timefrom ctypes
 import *from threading import Thread

# Load DLL

tdc = ctypes.CDLL(".\\Tdc_Libusb_Dll.dll")

# Define callback function (prints histogram maximum)

CALLBACK = CFUNCTYPE(None)def print_max():

    # print("hello")

    print(f"CH2 Max:{tdc.getHistoMax(2)}, CH3 Max:{tdc.getHistoMax(3)}")

c_callback = CALLBACK(print_max)

# Define function parameter types

tdc.dacwrite.argtypes = [c_int, c_int]

tdc.setuseroffset.argtypes = [c_int, c_int]

tdc.getHistoMax.argtypes = [c_int]

tdc.HistoT3Measure.argtypes = [ctypes.c_char_p, c_int, c_int, c_int,

                                POINTER(c_int), c_int, c_int, c_int, c_int, c_int, c_int, CALLBAC
K]

def main():

    """Main function"""

    # Initialize USB

    tdc.LibUsb_Init()

    # Set all channel DAC=500
    tdc.dacwrite(0, 500)
    tdc.dacwrite(1, 500)
    tdc.dacwrite(2, 500)
    tdc.dacwrite(3, 500)
    tdc.dacwrite(4, 500)
    tdc.dacwrite(5, 500)
    # Set channel offsets (1=-2000, others=0)
    offsets = [-2000, 0, 0, 0, 0, 0]
    for ch, val in enumerate(offsets, 1):
        tdc.setuseroffset(ch, val)
    # Start T3 measurement (channels 2 and 3, 1000 bins)
    channels = (c_int * 2)(2, 3)
    # filename = "";
    tdc.HistoT3Measure(b'',0, 10, 1, channels, 2, 0, 10, 1000, 3100, 500, c_callback)
    time.sleep(3.2) # Measure for 3.2 seconds
    tdc.stopHistoT3Measure() # Stop measurement
    # Get and print bin counts
    bins = (c_uint32 * 1000)()
    tdc.getBinCount(2, bins, 1000)
    for i, count in enumerate(bins):
        print(f"Bin{i}: {count}")
    tdc.LibUsb_Exit() # Release USB
if __name__ == "__main__":
    main()
```

4.9 Real-time Timestamp Acquisition Example (Python)

This example uses the ctypes library in a Windows Python environment to import the dynamic link library. It completes trigger threshold setting, channel delay compensation, and starts T2 measurement. Every 0.2 seconds, it acquires raw T2 timestamp data from the host buffer and writes it to a specified binary file. The complete source code is as follows:

python

```
#coding=gb2312# Single Photon Counter DLL Control and Data Acquisition Example# In Windows, use Python to call the dynamic link library, implement timestamp measurement, acquire measurement data in real-time, and write to a specified file.import ctypes;from ctypes import *import time

# Import dynamic link library from local directory

tDll = CDLL("./Tdc_Libusb_Dll.dll");print('Hello SIMTURM!')# Declare a buffer, size 32MB, can store 4M timestamps. For high count rates or larger measurement intervals, increase buffer size.# Use ctypes c_char type

dBuffer = (c_char*32*1024*1024)()# Initialize USB environment

tDll.LibUsb_Init()# write threshold voltage, start from 0 to 5for ch in range(0,6):

    tDll.dacwrite(ch, 500)# write channel offset, start from 1 to 6for ch in range(1,7):

    tDll.setuseroffset(ch, 0)

totalbytes = 0# Write to tstd.bin in current directoryfile = open('tstd.bin','wb')# Start measurement, timestamp mode

tDll.start_tstd_mmf_usb(0)# Wait 0.2 seconds

time.sleep(0.2)# Read measurement data.# Param 1: Maximum number of bytes to read is buffer size.# Param 2: Destination address to copy data.# Return value: Actual number of bytes read.

rbytes = tDll.get_tstd_mmf_usb_data(32*1024*1024, dBuffer)# Convert buffer input to byte array

bytes2w = ctypes.string_at(dBuffer, rbytes)

totalbytes += rbytes;# Write to filefile.write(bytes2w)# Wait 0.2 seconds

time.sleep(0.2)

rbytes = tDll.get_tstd_mmf_usb_data(32*1024*1024, dBuffer)

bytes2w = ctypes.string_at(dBuffer, rbytes)file.write(bytes2w)

totalbytes += rbytes# Stop measurement

tDll.stop_tstd_mmf_usb();# Read residual data and write to file (can also be ignored)

rbytes = tDll.get_tstd_mmf_usb_data(32*1024*1024, dBuffer)

bytes2w = ctypes.string_at(dBuffer, rbytes)file.write(bytes2w)

totalbytes += rbytes# Close memory-mapped file (must)

tDll.close_tstd_mmf()# Close filefile.flush()file.close()print("Measurement Done! " + str(totalbytes) + " bytes are written")# Exit USB environment
```

tDll.LibUsb_Exit();

4.10 Photon Counting Statistics Example (Sync Signal Interval Mode, Python)

This example demonstrates how to call the DLL in Python, designate a channel's signal (example uses channel 1) as the interval signal, and count the number of photons on other channels within the sync signal intervals.

The user calls the start_irrt_mmf function to start photon counting statistics and stop_irrt_mmf() to stop intensity monitoring. During measurement or after stopping, get_irrt_mmf_usb_data can be called to get photon statistics for all channels. In the example, the acquired photon count records are written to a file in binary format. Before exiting the program, close_irrt_mmf must be called to release resources.

The main function int start_irrt_mmf(int sync, int window_len) has two parameters. The first parameter sync is the channel number where the sync signal resides. When sync equals 1~6, it indicates measurement in Sync Signal mode, and the second parameter is ignored. If sync equals 0, then measurement is in Time Window mode, and the time window length is specified by the second parameter window_len, unit us.

In the example, start_irrt_mmf is first called to start measurement, then photon count data is read every 0.2 seconds and written to a file. After 10 data reads, stop_irrt_mmf is called to stop measurement. Remember to call close_irrt_mmf() to release resources before exiting the program. The saved data can be parsed using the provided Matlab script "proc_intense_sync.m". Note that the script needs to specify the sync channel, and the part parsing the header should be commented out (the host software saved file includes a header, while data collected with this program does not have a header).

The Python source code is provided below for reference:

python

```
#coding=gb2312# Single Photon Counter DLL Control and Data Acquisition Example# In Windows, use Python to call the dynamic link library, implement timestamp measurement, acquire measurement data in real-time, and write to a specified file.import ctypes;from ctypes import *import time# Import dynamic link library from local directory

tDll = CDLL("./Tdc_Libusb_Dll.dll");print('Hello SIMTURM!')# Declare a buffer, size 32MB, can store 4M timestamps. For high count rates or larger measurement intervals, increase buffer size.# Use ctypes c_char type

dBuffer = (c_char*32*1024*1024)()# Initialize USB environment

tDll.LibUsb_Init()# write threshold voltage, start from 0 to 5for ch in range(0,6):

    tDll.dacwrite(ch, 100)# write channel offset, start from 1 to 6for ch in range(1,7):
```

```
    tDll.setuseroffset(ch, 0)

totalbytes = 0# Write to irrt.bin in current directoryfile = open('irrt.bin','wb')# Start measurement, sync channel is 1, 200 ignored (since sync mode)

tDll.start_irrt_mmf(1,200);for i in range(1,10):

    # Wait 0.2 seconds

    time.sleep(0.2)

    # Read measurement data.

    # Param 1: Maximum number of bytes to read is buffer size.

    # Param 2: Destination address to copy data.

    # Return value: Actual number of bytes read.

    rbytes = tDll.get_irrt_mmf_usb_data(32*1024*1024, dBuffer)

    # Convert buffer input to byte array

    bytes2w = ctypes.string_at(dBuffer, rbytes)

    totalbytes += rbytes;

    # Write to file

    file.write(bytes2w)

    print('%d turn finished' %i)# Stop measurement

tDll.stop_irrt_mmf();# Read residual data and write to file (can also be ignored)

rbytes = tDll.get_irrt_mmf_usb_data(32*1024*1024, dBuffer)

bytes2w = ctypes.string_at(dBuffer, rbytes)file.write(bytes2w)

totalbytes += rbytes# Close memory-mapped file (must)

tDll.close_irrt_mmf()# Close filefile.flush()file.close()print("Measurement Done! " + str(totalbytes) + " bytes are written")# Exit USB environment

tDll.LibUsb_Exit();
```

4.11 Photon Counting Statistics (Time Window Mode, C++)

This example demonstrates how to perform photon counting statistics in Time Window mode in a C++ environment. Like the previous example, it uses the `start_irrt_mmf` function, but the first parameter is set to 0, indicating Time Window mode. The second parameter is the window length, unit us, range 1~8500000 (8.5 seconds).

The example sets the window length to 1 second and reads photon statistics data every 1 second, printing the count rate. Measurement ends after 5 seconds. close_irrt_mmf is called to release resources.

Note that the internal photon count buffer size is 64 MB. With a window length of 1 us, data will overflow after 1.39 seconds. Therefore, if the window is 1 us, try to keep the data acquisition interval less than or equal to 1 second.

The source code is provided below for reference:

cpp

```cpp
#include "Tdc_Libusb_Dll.h"#include <iostream>#define CHN_CNT (6)#define BUF_SIZE (1024*1024)char destBuffer1[BUF_SIZE];int main(){

    std::cout << "Hello World!\n";

    // Initialize USB environment

    LibUsb_Init();

    printf("usb init done!\r\n");

    // Set threshold voltage 500mV

    for (int i = 0; i < CHN_CNT; i++)

    {

        dacwrite(i, 500);

    }

    printf("set threshold voltage done!\r\n");

    // Set channel delays

    setuseroffset(1, 0);

    setuseroffset(2, 0);

    setuseroffset(3, 0);

    setuseroffset(4, 0);

    setuseroffset(5, 0);

    setuseroffset(6, 0);

    int interval_us = 1000000; // Count photons per second

    start_irrt_mmf(0, interval_us);

    Sleep(10);
    for (int i = 0; i < 5; i++)

    {

        Sleep(1000);

        int bytes = get_irrt_mmf_usb_data(BUF_SIZE, destBuffer1);

        for (int i = 0; i < CHN_CNT; i++)

        {

            UINT64 val = *(UINT64*)(destBuffer1 + (i * 8));

            UINT32 chan = val >> 32;

            UINT32 rate_uint = val & 0xFFFFFFFF;

            float rate = rate_uint / (interval_us / 1e6);

            printf("hitrate of channel %d is %f\r\n", chan, rate);

        }

    }

    stop_irrt_mmf();

    // Release resources (must)

    close_irrt_mmf();

    LibUsb_Exit();}
```

5 Appendix

5.1 Timestamp Format

One timestamp record is 64 bits, 8 bytes, format as follows:

Bits [63:57]	**Bits [56:0]**
Unsigned integer, Channel Number	Timestamp, signed integer, LSB unit: ps

5.2 Time Difference Format

When the channel is the sync signal:

Bits [63:57]	**Bits [56:0]**

Bits [63:57]	Bits [56:0]
Unsigned integer, Channel Number	Timestamp, signed integer, unit: ps

When the channel is not the sync signal:

Parsing the time difference file, each 8 bytes represent a time difference record, format as follows:

Bits [63:57]	Bits [56:0]
Unsigned integer, Channel Number	Time difference relative to sync, ps

5.3 Single Photon Intensity Format

One intensity record is 64 bits, 8 bytes, format as follows:

Bits [63:40]	Bits [39:32]	Bits [31:0]
Sequence number of the corresponding sync pulse	Channel Number	Count Value

The first record is the header, format as follows:

Bits [63:40]	Bits [39:32]	Bits [31:0]
Sync signal channel number (sync)	0x00	Time window length (us)

Matlab Script List

No.	File Name	Function
1	proc_histogram.m	Parses real-time histogram saved files.
2	proc_intense_sync.m	Parses photon intensity measurement saved files.
3	proc_t3_64b.m	Parses T3 mode saved files.
4	proc_ts_sfp	Parses 10 Gigabit optical port timestamp saved files.
5	proc_ts_usb	Parses USB timestamp saved files.
6	proc_histscanlog	Parses T2 scan histogram saved histogram data.

[^1]: As long as the measurement does not span the counter rollover moment, compensation is unnecessary. If long-duration continuous measurement is required and might span a rollover cycle, please contact our company for data processing methods.