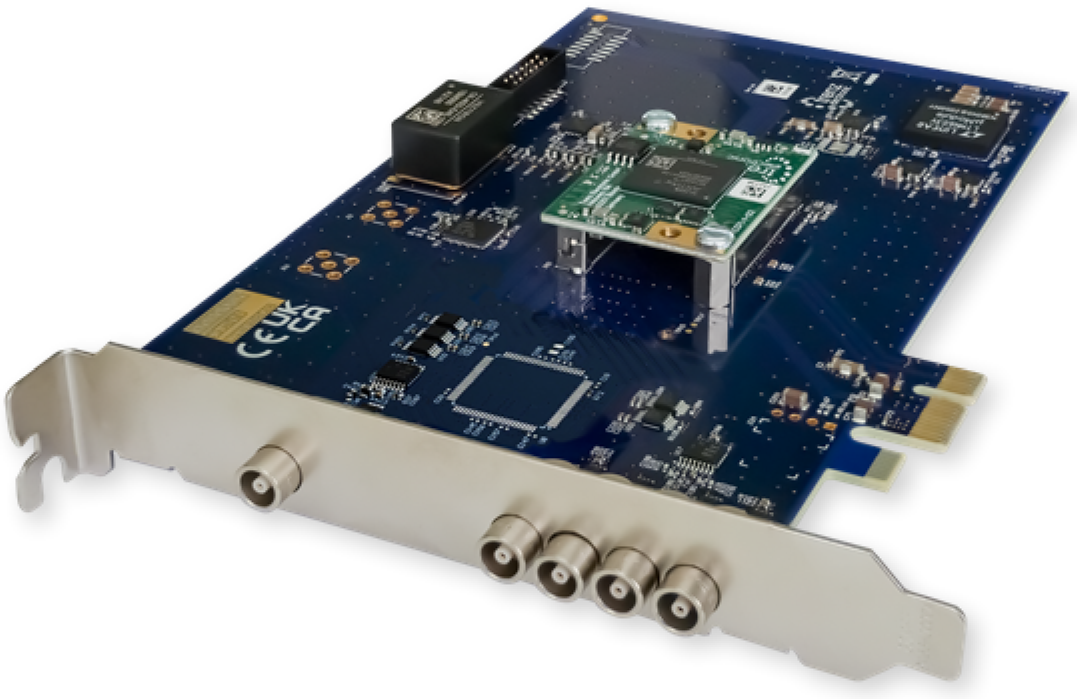
cronologic

TimeTagger

TimeTagger4

User Guide



www.cronologic.de

Contents

1 Introduction

The TimeTagger4 is a *common-start* low resolution high throughput time-to-digital converter. Timestamps of leading or trailing edges (or both) of digital pulses are recorded. The TimeTagger4 produces a stream of output packets, each containing data from a single start event. The relative timestamps of all stop pulses that occur within a configurable range are grouped into one packet.

1.1 Features

- 4-channel common-start TDC
- Quantization (measurement resolution): 100 ps to 1000 ps
- Standard Range: 8.388 ms for Gen 1 and 1.677 ms for Gen 2 (24 bits)
- Extended Range: 2.147 s for Gen 1 and 0.429 s for Gen 2 (31 bits)
- Double-pulse resolution: twice the quantization size
- Dead time between groups: none
- Minimum interval between starts: 4 ns for Gen 1 and 3.2 ns for Gen 2
- Up to 8000 Hits per Packet
- 5 to 0.625 GHz/s for bursts of up to 4096 starts
- 5 to 0.625 GHits/s per channel for bursts of up to 3900 stops
- 40 MHits/s per channel of sustained stops
- 60 MHits/s over all channels of sustained stops
- PCIe x1 interface
- Continuous Mode supporting unlimited recording of stop signal streams even without start signals

The TimeTagger4 exists in different variants and resolutions from TimeTagger4-1G to TimeTagger4-10G.

Parameter	-1G	-2G	-1.25G	-2.5G	-5G	-10G
Quantization	1000 ps	500 ps	800 ps	400 ps	200 ps	100 ps
Data Format Bin Size	500 ps	500 ps	100 ps	100 ps	100 ps	100 ps
DNL and INL	0.01 LSB	0.01 LSB	0.01 LSB	0.01 LSB	0.01 LSB	0.3 LSB
PCIe line rate	Gen 1	Gen 1	Gen 2	Gen 2	Gen 2	Gen 2
Readout Rate	200 MB/s	200 MB/s	400 MB/s	400 MB/s	400 MB/s	400 MB/s
Status	end of life	end of life	active	active	active	active

1.2 Applications

The TimeTagger4 can be used in all time measurement applications where a common-start setup with 100 ps resolution is sufficient. For alternatives with higher resolution, more channels or higher readout rates check our TDC website www.cronologic.de.

The TimeTagger4 is well suited for the following applications:

- LIDAR down to 8 cm resolution
- blade oscillation measurements
- reciprocal counters
- coincidence measurements
- quantum key distribution (QKD)
- time correlated single photon counting (TCSPC)

2 Hardware

The TimeTagger4 is available as a PCIe plugin board (variant "-PCIe", see Ordering Information) or as a desktop solution (variant "-TBT"). The two variants are shown in Figure 2.1.

For a detailed overview of the hardware of the TBT-variant, please refer to our respective user guide.

Figure 2.1 Overview of the PCIe (left) and the TBT (right) variant of the TimeTagger4.

2.1 Installing the PCIe Board

The TimeTagger4 board can be installed in any PCIe-CEM slot with x1 or more lanes. Make sure that the PC is powered off and that the main power connector is disconnected while installing the board.

2.2 TimeTagger4 Inputs and Connectors

Figure 2.2 shows the location of the inputs on the slot bracket.

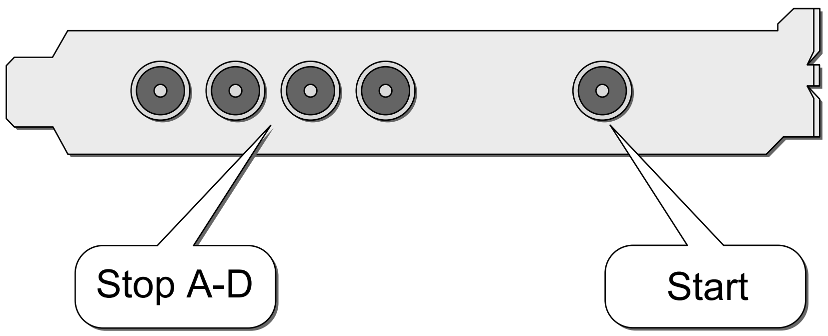


Figure 2.2 Input connectors of the TimeTagger4 on the PCIe bracket.

LEMO-00 connectors are used for input connection. The inputs are AC-coupled and have an impedance of 50 Ω. A schematic of the input circuit is shown in Figure 2.3. The digital threshold for any input can be

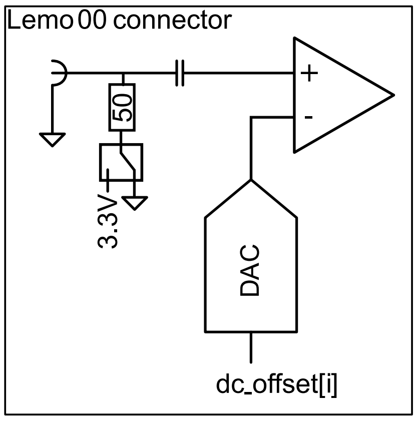


Figure 2.3 Input circuit for each of the input channels.

adjusted to comply with a multitude of single-ended signaling standards. The threshold can also be used to configure the input for either positive or negative pulses.

The connectors can also be used as outputs. DC-coupled output pulses for automatic internal triggering and control of external devices can be generated using the TiGer timing pattern generator. See Section 3.5 for details on the TiGer. Furthermore, for **Gen 1** boards three inter-board connectors can be found near the top edge of the TimeTagger4 board, as displayed in Figure 2.4. Connector J25 is reserved for future use. The pinout of connector J12 is shown in Table 2.1 and the pinout of connector J6 is depicted in Table 2.2. **Gen 2 boards do not possess these three connectors**.

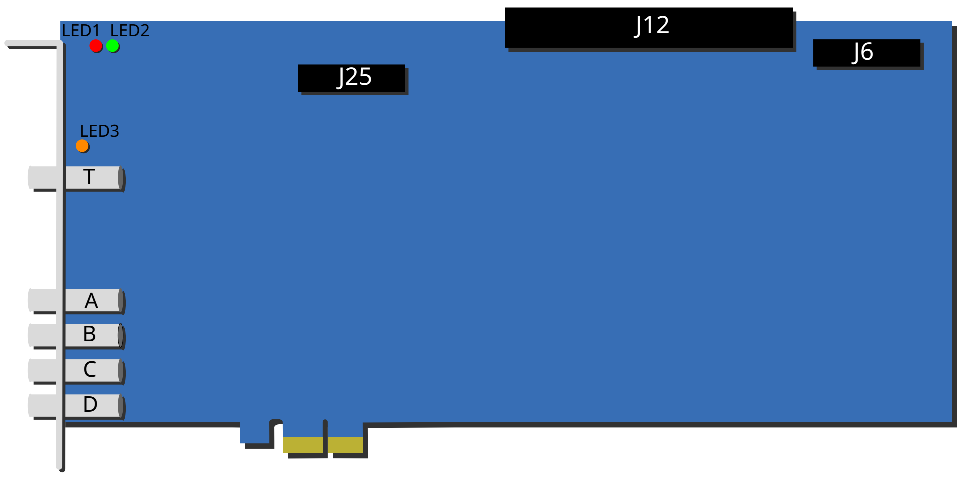


Figure 2.4 Schematic view of a TimeTagger4 board including the inter-board connectors. The inter-board connectors are only present on Gen 1 boards.

Pin	Name
1, 2	GND
3, 4	external CLK in N, external CLK in P
5, 6	GND
7, 8	reserved/NC
9, 10	GND
11, 12	reserved/NC
13, 14	GND
15, 16	reserved/NC
17, 18	GND
19, 20	reserved/NC
21, 22	GND
23, 24	reserved/NC
25, 26	GND
27, 28	reserved/NC
29, 30	GND
31, 32	reserved/NC
33, 34	GND

Table 2.1 Pinout of connector J12.

Pin	Name
1	+3.3 V
2 - 9	reserved/NC
10	GND

Table 2.2 Pinout of connector J6.

2.3 Status LEDs of the PCIe boards

Three status LEDs are present on the board, as seen in Figure 2.4.

- LED1 lights up red during the configuration of the FPGA and turns off afterward. If it stays permanently lit, the configuration failed.
- LED2 lights up green after the board is initialized by the driver and turns off when the device is closed by the software.
- LED3 lights up green when capture is started, yellow after the first start signal was detected and red when groups are missing.

3 TimeTagger4 Functionality

The TimeTagger4 is a “classic” common start time-to-digital converter.

It records the time difference between a leading or trailing edge on the start input to the leading and trailing edges of the stop inputs. Rising and falling edges of the stop channels A-D can be enabled individually. The measurements are quantized as shown in Section 1.1. The timestamps are recorded in integer multiples of the corresponding bin size. Transitions of the input signals are called hits. To reliably detect hits the signal has to be stable for more than one quantization interval before and after the edge. Triggers on the start channel must not occur less than 5 ns apart. The TimeTagger4 records events without dead time at a readout rate of about 48 MHits/s for Gen 1 and 60 MHits/s for Gen 2. For Gen 2, the maximum readout rate of a single channel is 40 MHits/s.

3.1 Grouping and Events

In typical applications a start hit is followed by a multitude of stop hits. If grouping is enabled, the hits recorded are managed in groups (which are called “events” in some applications).

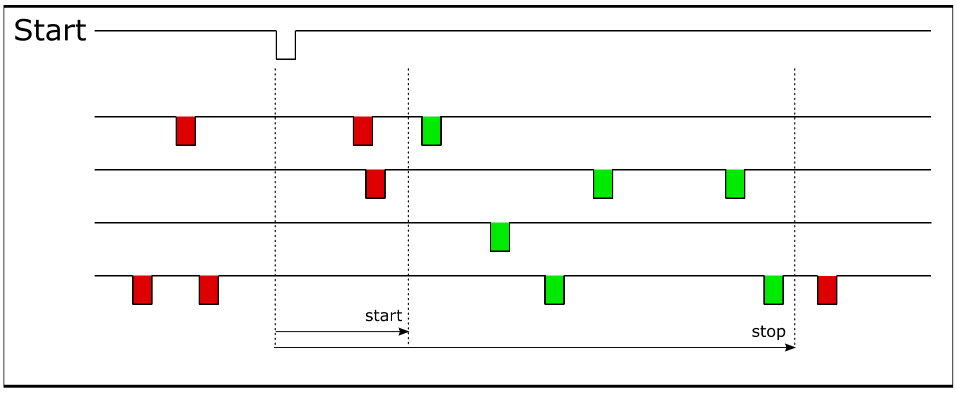


Figure 3.1 Acquired hits are merged to groups as explained in the text.

Figure 3.1 shows a corresponding timing diagram. The user can define the range of a group, i.e., the time window within which hits on the stop channels are recorded. Hits occurring outside that time window are discarded.

Different ranges can be set for each of the stop channels by setting corresponding values for `channel[i].start` and `channel[i].stop` values.

The values need to be set as multiples of the bin size and must not be negative.

$$0 \leq start \leq stop \leq 2^{16} - 1$$

If a second start is recorded within the range of a group, the current group is finished and a new group is started. Consecutive stops will be assigned to the new group (as long as they are within the group range).

3.2 Auto-Triggering Function Generator

Some applications require internal periodic or random triggering. The TimeTagger4 function generator provides this functionality.

The delay between two trigger pulses of this trigger generator is the sum of two components: A fixed value M and a pseudo-random value with a range given by the exponent N.

The period is

$$T = M + [1...2^N] - 1$$

clock cycles with a duration of 4 ns per cycle for Gen 1 and 3.2 ns for Gen 2 TimeTagger4. The standard values of $M = 62500$ and $N = 0$ result in a frequency of 4 kHz for TimeTagger4 Gen 1 and 5 kHz for Gen 2 devices. Here, $M_{\text{min}} \leq M < 2^{32}$ and $0 \leq N < 32$, where $M_{\text{min}} = 6$ for Gen 1 and $M_{\text{min}} = 8$ for Gen 2.

The trigger can be used as a source for the TiGer unit (see Section 3.5) and defines the period for the continuous mode (see Section 3.3).

3.3 Continuous Mode

This feature is only available for Gen 2 devices of the TimeTagger4.

The TimeTagger4 continuously records stop signals even without a start signal connected. The data stream contains periodic packets with an absolute timestamp of 64 bits, followed by a list of stops relative to this timestamp. The period of the timestamps can be adjusted using the Auto-Triggering Function Generator (see Section 3.2) to adapt them to your evaluation interval. Lower frequencies will create larger packets and have therefore a larger latency for receiving packets, potentially overflowing the buffers. Frequencies lower or equal to 600 Hz will contain rollover. Apart from that the choice is arbitrary.

3.4 Configurable Input Delay

This feature is only available for Gen 2 devices of the TimeTagger4.

Each of the five input channels of the TimeTagger4 can be delayed for up to 204.6 ns with a 200 ps granularity.

3.5 Timing Generator (TiGer)

Each digital LEMO-00 input can be used as an LVCMOS trigger output. The TiGer functionality can be configured independently for each connector. See Section 4.5.3 for a full description of the configuration options.

Figure 3.2 shows how the TiGer blocks are connected. They can be triggered by an OR of an arbitrary combination of inputs, including the auto-trigger . Each TiGer can drive its output to its corresponding LEMO connector. This turns the connector into an output.

The TiGer is DC coupled to the connector. Connected hardware must not drive any signals to connectors used as outputs, as doing so could damage both the TimeTagger4 and the external hardware. Pulses that are short enough for the input AC coupling are available as input signals to the TimeTagger4. This can be used to measure exact time differences between the generated output signals and input signals on other channels. When using one of the input channels as a source for the TiGer, the expected latency between signal input and TiGer output is roughly 95 ns.

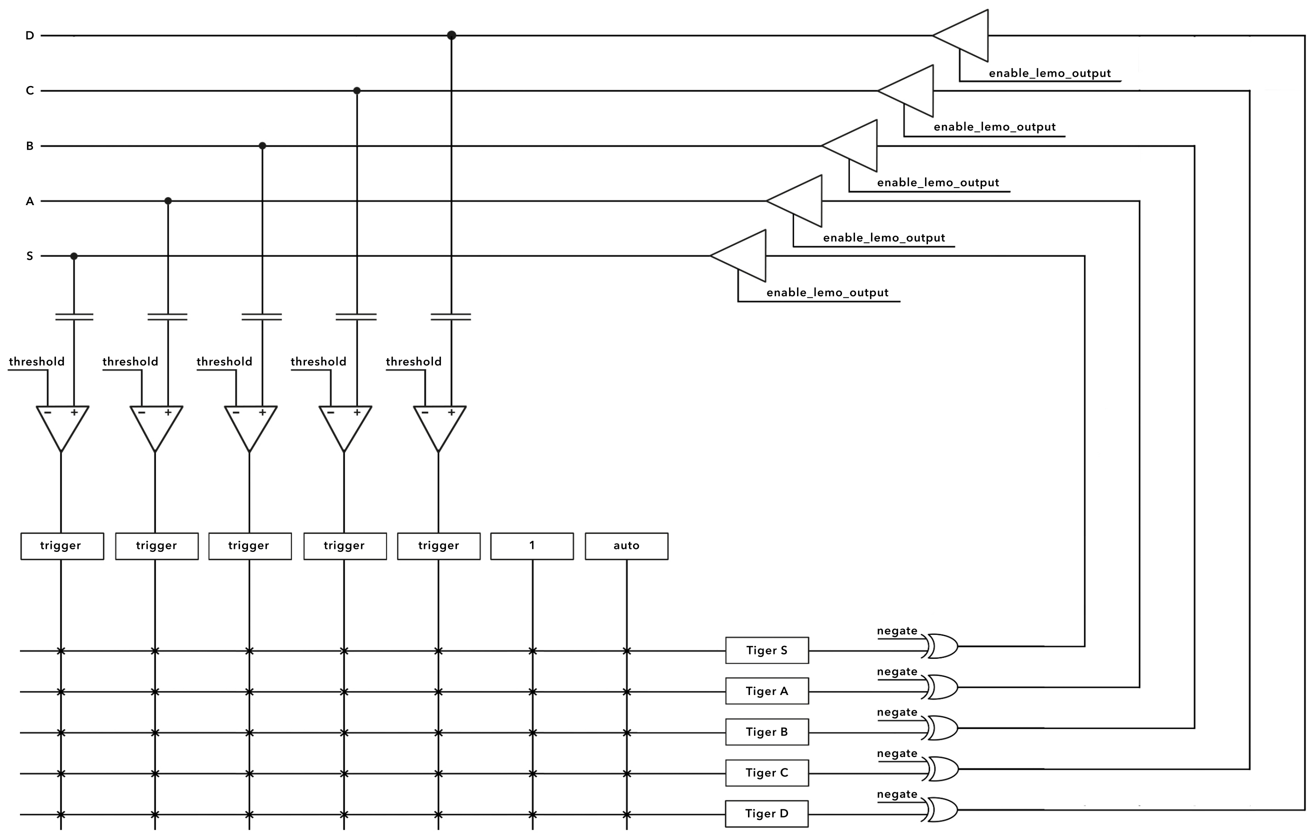


Figure 3.2 TiGer blocks can generate outputs that are also available on inputs.

3.6 Calibration of the 10G variant

After performing a firmware update, the TimeTagger4-10G has to be re-calibrated. This does *only* apply to the 10G variant.

For this purpose, cronologic provides the `timetagger4_10g_calibration_64` command-line tool (located in the driver installation directory in `apps/x64`).

To perform a calibration, you need a NIM signal with a constant frequency larger than 20 kHz and smaller than 15 MHz.

Run the `timetagger4_10g_calibration_64` tool and follow the instructions on-screen, that is:

- Connect the NIM signal to channel Start and press enter. The tool will calibrate the Start channel.
- After Start was successfully calibrated, connect the same NIM signal to channel A and press enter. The tool will calibrate channel A.
- Repeat this process for channels B, C, and D, as well.

After a successful calibration of the TimeTagger4-10G, the message "Calibrated all channels successfully" will be displayed. You can close the calibration tool.

In case the calibration fails, please check the following:

- The TimeTagger4-10G is installed properly (see Section 2).
- A proper NIM signal with a constant frequency larger than 20 kHz and smaller than 10 MHz is used.
- The NIM signal was connected to the appropriate input channel (see Figure 2.2).

If the calibration still fails, please contact cronologic support.

4 Driver Programming API

The API is a DLL with C linkage.

The functions provided by the driver are declared in `TimeTagger4_interface.h` which must be included by your source code. You must tell your compiler to link with the file xTDC4_driver_64.lib. When running your program the dynamic link library containing the actual driver code must reside in the same directory as your executable or be in a directory included in the PATH variable. For Linux, it is provided only as a static library `libxtdc4_driver.a` The file for the DLL is called `xTDC4_driver_64.dll`.

All these files are provided with the driver installer that can be downloaded from the product website www.cronologic.de. By default, the installer will place the files into the directory `C:\Program Files\cronologic\TimeTagger4\driver`. A coding example can be found on github.com/cronologic-de/xtdc_babel.

4.1 Constants

`#define` **`TIMETAGGER4_TDC_CHANNEL_COUNT 4`**
The number of TDC input channels.

`#define` **`TIMETAGGER4_TIGER_COUNT 5`**
The number of timing generators. One for each TDC input and one for the start input.

`#define` **`TIMETAGGER4_TRIGGER_COUNT 16`**
The number of potential trigger sources for the timing generators. One for each TDC input, one for the Start input plus some specials. See Section 4.5.3 for details.

`#define` **`TIMETAGGER4_OK 0`**
Error codes are set by the API functions to this value if there has been no error. Other error codes can be found in `TimeTagger4_interface.h`

4.2 Driver Information

Even if there is no board present the driver revision can be queried using these functions.

`int` **`timetagger4_get_driver_revision()`**
Returns the driver version, same format as `timetagger4_static_info.driver_revision`. This function does not require a TimeTagger4 board to be present.

`const char*` **`timetagger4_get_driver_revision_str()`**
Returns the driver version including SVN build revision as a string.

`int` **`timetagger4_count_devices(`**`int` **`*error_code,`** `char` **`**error_message)`**
Returns the number of boards present in the system that are supported by this driver. Pointers to an error code and message variable have to be provided. If `error_code` does not equal `#define TIMETAGGER4_OK = 0`, the error message will contain what went wrong. E.g., the crono kernel was not properly installed.

4.3 Initialization

The card must be initialized first before reading data. Normally the process is to get the default initialization parameters and change some values. E.g., choose one of multiple cards by the index or use a larger buffer.

`int timetagger4_get_default_init_parameters(timetagger4_init_parameters *init)`

Sets up the standard parameters. Gets a set of default parameters for `timetagger4_init()`. This must always be used to initialize the `timetagger4_init_parameters` structure before modifying it and passing it to `timetagger4_init`.

`timetagger4_device timetagger4_init(timetagger4_init_parameters *params, int *error_code, char **error_message)`

Opens and initializes the TimeTagger4 board with the given index.

`error_code` must point to an integer where the driver can write the error code.

`error_message` must point to a pointer to char. The driver will allocate a buffer for zero-terminated error message and store the address of the buffer in the location provided by the user.

The parameter `params` is a pointer to a structure of type `timetagger4_init_parameters` that must be completely initialized by `get_default_init_parameters()`.

`int timetagger4_close(timetagger4_device *device)`

Closes the devices, releasing all resources.

4.3.1 Structure timetagger4_init_parameters

`int version`

The version number. Must be set to `TIMETAGGER4_API_VERSION`.

`int card_index`

The index in the list of TimeTagger4 boards that should be initialized.

There might be multiple boards in the system that are handled by this driver as reported by `timetagger4_count_devices`. This index selects one of them. Boards are enumerated depending on the PCIe slot. The lower the bus number and the lower the slot number the lower the card index.

`int board_id`

The global index in all cronologic devices.

This 8-bit number is filled into each packet created by the board and is useful if data streams of multiple boards will be merged. If only TimeTagger4 cards are used this number can be set to the `card_index`. If boards of different types that use a compatible data format are used in a system each board should get a unique ID. Can be changed with `int timetagger4_set_board_id (timetagger4_device *device, int board_id)`.

`uint64_t buffer_size[8]`

The minimum size of the DMA buffer.
If set to 0 the default size of 16 MByte is used. For the TimeTagger4, only the first entry is used.

`int buffer_type`

The type of buffer. Must be set to 0.

```
#define TIMETAGGER4_BUFFER_ALLOCATE 0
```

`#define TIMETAGGER4_BUFFER_USE_PHYSICAL 1` // unsupported

`uint64_t` **`buffer_address`**

This is set by `timetagger4_init()` to the start address of the reserved memory.

The buffers will be allocated with the sizes given above. Make sure that the memory is large enough.

`int` **`variant`**

Set to 0. Can be used to activate future device variants such as different base frequencies.

`int` **`device_type`**

A constant for the different devices of cronologic `CRONO_DEVICE_*`.

Initialized by `timetagger4_get_default_init_parameters()`. This value is identical to the PCI Device ID. Must be left unchanged.

```
#define CRONO_DEVICE_HPTDC        0x1
#define CRONO_DEVICE_NDIGO5G      0x2
#define CRONO_DEVICE_NDIGO250M    0x4
#define CRONO_DEVICE_xTDC4        0x6
#define CRONO_DEVICE_TIMETAGGER4  0x8
#define CRONO_DEVICE_XHPTDC8      0xC
#define CRONO_DEVICE_NDIGO6       0xD
```

`int` **`dma_read_delay`**

The update delay of the write pointer after a packet has been sent over PCIe. Specified in multiples of 16 ns. Should not be changed by the user.

`int` **`use_ext_clock`**

If set to 1, use external 10 MHz reference. If set to 0, use internal reference.

4.4 Status Information

4.4.1 Functions for Information Retrieval

The driver provides functions to retrieve detailed information on the board type, its configuration, settings, and state. The information is split according to its scope and the computational requirements to query the information from the board.

`int` **`timetagger4_get_device_type(`**`timetagger4_device` **`*device)`**

Returns the type of the device as `CRONO_DEVICE_TIMETAGGER4`.

`const char*` **`timetagger4_get_last_error_message(`**`timetagger4_device` **`*device)`**

Returns most recent error message.

`int` **`timetagger4_get_fast_info(`**`timetagger4_device` **`*device,`** `timetagger4_fast_info` **`*info)`**

Returns fast dynamic information.

This call gets a structure that contains dynamic information that can be obtained within a few microseconds.

`int timetagger4_get_param_info(timetagger4_device *device, timetagger4_param_info *info)`

Returns configuration changes.

Gets a structure that contains information that changes indirectly due to configuration changes.

`int timetagger4_get_static_info(timetagger4_device *device, timetagger4_static_info *info)`

Contains static information.

Gets a structure that contains information about the board that does not change during run time.

`int timetagger4_get_pcie_info(timetagger4_device *device, crono_pcie_info *pcie_info)`

Read PCIe information.

Gets a structure that contains information about the PCIe state, like correctable or uncorrectable errors.

`int timetagger4_clear_pcie_errors(timetagger4_device *device, int flags)`

Clear PCIe errors.

Only useful for PCIe debugging. `flags` is one of the following:

```
#define CRONO_PCIE_CORRECTABLE_FLAG     1

#define CRONO_PCIE_UNCORRECTABLE_FLAG   2
```

4.4.2 Structure timetagger4_static_info

This structure contains information about the board that does not change during run time. It is provided by the function `timetagger4_get_static_info()`.

`int size`

The number of bytes occupied by the structure.

`int version`

A version number that is increased when the definition of the structure is changed. The increment can be larger than one to match driver version numbers or similar.

`int board_id`

ID of the board.

`int driver_revision`

Encoded version number for the driver.

The lower three bytes contain a triple-level hierarchy of version numbers, e.g., `0x010103` encodes version 1.1.3.

The version adheres to the Semantic Versioning scheme as defined at https://semver.org. A change in the first digit generally requires a recompilation of user applications. Changes in the second digit denote significant improvements or changes that don't break compatibility and the third digit increments with minor bug fixes and similar updates that do not affect the API.

`int driver_build_revision`

Build number of the driver according to cronologic's internal versioning system.

`int firmware_revision`

Revision number of the FPGA configuration.

`int` **`board_revision`**

Board revision number.

The board revision number can be read from a register. It is a four-bit number that changes when the schematic of the board is changed. This should match the revision number printed on the board.

`int` **`board_configuration`**

Describes the schematic configuration of the board.

The same board schematic can be populated in multiple variants. This is an 8-bit code that can be read from a register.

`int` **`subversion_revision`**

Subversion revision ID of the FPGA configuration source code.

`int` **`chip_id`**

Reserved.

`int` **`board_serial`**

Serial number.

Year and running number are concatenated in 8.24 format. The number is identical to the one printed on the silvery sticker on the board.

`unsigned int` **`flash_serial_high`**
`unsigned int` **`flash_serial_low`**

64-bit manufacturer serial number of the flash chip

`crono_bool_t` **`flash_valid`**

If not 0, the driver found valid calibration data in the flash on the board and is using it. This value is not applicable for the TimeTagger4.

`char` **`calibration_date[20]`**

DIN EN ISO 8601 string YYYY-MM-DD HH:MM of the time when the card was calibrated.

`char` **`bitstream_date[20]`**

DIN EN ISO 8601 string YYYY-MM-DD HH:MM of the time when the bitstream on the card was created.

`double` **`delay_bin_size`**

Bin size of delay in picoseconds. The increment of the `delay_config.delay` field for the TimeTagger4.

`double` **`auto_trigger_ref_clock`**

Auto trigger clock frequency. The clock frequency of the auto trigger in Hertz used for calculating the `auto_trigger_period`.

`uint32_t` **`rollover_period`**

The number of bins in a rollover period. This is a power of 2 (the maximum value of a hit timestamp is this value minus 1)

4.4.3 Structure timetagger4_param_info

This structure contains configuration changes provided by `timetagger4_get_param_info()`.

`int` **`size`**

The number of bytes occupied by the structure.

`int` **`version`**

A version number that is increased when the definition of the structure is changed. The increment can be larger than one to match driver version numbers or similar.

`double` **`binsize`**

Bin size (in ps) of the measured TDC data.

`int` **`board_id`**

Board ID.

The board uses this ID to identify itself in the output data stream. The ID takes values between 0 and 255.

`int` **`channels`**

Number of TDC channels of the board.

Fixed at 4.

`int` **`channel_mask`**

Bit assignment of each enabled input channel.

Bit $0 \leq n < 4$ is set if channel n is enabled.

`int64_t` **`total_buffer`**

The total amount of DMA buffer in bytes.

`double` **`packet_binsize`**

For the TimeTagger4 the packet binsize is equal to the binsize and depends on the generation of the card. Gen 1 boards have a packet binsize of 500 ps, while Gen 2 boards have 100 ps.

`double` **`quantisation`**

Quantisation or measurement resolution. Depending on the board variant this ranges from 100 ps to 1000 ps.

–1G	–2G	–1.25G	–2.5G	–5G	–10G
1000 ps	500 ps	800 ps	400 ps	200 ps	100 ps

This means, that for –1.25G the lower 3 bits in the timestamps are zero.

4.4.4 Structure timetagger4_fast_info

`int` **`size`**

The number of bytes occupied by the structure.

`int` **`version`**

A version number that is increased when the definition of the structure is changed. The increment can be larger than one to match driver version numbers or similar.

`int` **`tdc_rpm`**

Speed of the TDC fan in rounds per minute. Reports 0 if no fan is present.

`int` **`fpga_rpm`**

Speed of the FPGA fan in rounds per minute. Reports 0 if no fan is present.

`int` **`alerts`**
: Alert bits from the temperature sensor and the system monitor. The TimeTagger4 does not implement any temperature alerts.

`int` **`pcie_pwr_mgmt`**
: Always 0.

`int` **`pcie_link_width`**
: Number of PCIe lanes the card uses. Should always be 10 for the TimeTagger4.

`int` **`pcie_max_payload`**
: Maximum size in bytes for one PCIe transaction. Depends on system configuration.

4.4.5 Structure crono_pcie_info

`uint32_t` **`pwr_mgmt`**
: Organizes power supply of PCIe lanes.

`uint32_t` **`link_width`**
: Number of PCIe lanes that the card uses.

`uint32_t` **`max_payload`**
: Maximum size in bytes for one PCIe transaction.

Depends on the system configuration.

`uint32_t` **`link_speed`**
: Data rate of the PCIe card.

Depends on the system configuration.

`uint32_t` **`error_status_supported`**
: Different from 0 if the PCIe error status is supported for this device.

`uint32_t` **`correctable_error_status`**
: Correctable error status flags, directly from the PCIe config register.

Useful for debugging PCIe problems.

```
#define CRONO_PCIE_RX_ERROR                  1 << 0
#define CRONO_PCIE_BAD_TLP                   1 << 6
#define CRONO_PCIE_BAD_DLLP                  1 << 7
#define CRONO_PCIE_REPLAY_NUM_ROLLOVER       1 << 8
#define CRONO_PCIE_REPLAY_TIMER_TIMEOUT      1 << 12
#define CRONO_PCIE_ADVISORY_NON_FATAL        1 << 13
#define CRONO_PCIE_CORRECTED_INTERNAL_ERROR  1 << 14
#define CRONO_PCIE_HEADER_LOG_OVERFLOW       1 << 15
```

`uint32_t` **`correctable_error_status`**
: Uncorrectable error status flags, directly from the PCIe config register.

Useful for debugging PCIe problems.

```
#define CRONO_PCIE_UNC_UNDEFINED                    1 << 0
#define CRONO_PCIE_UNC_DATA_LINK_PROTOCOL_ERROR     1 << 4
#define CRONO_PCIE_UNC_SURPRISE_DOWN_ERROR          1 << 5
#define CRONO_PCIE_UNC_POISONED_TLP                 1 << 12
#define CRONO_PCIE_UNC_FLOW_CONTROL_PROTOCOL_ERROR  1 << 13
#define CRONO_PCIE_UNC_COMPLETION_TIMEOUT           1 << 14
#define CRONO_PCIE_UNC_COMPLETER_ABORT              1 << 15
#define CRONO_PCIE_UNC_UNEXPECTED_COMPLETION        1 << 16
#define CRONO_PCIE_UNC_RECEIVER_OVERFLOW_ERROR      1 << 17
#define CRONO_PCIE_UNC_MALFORMED_TLP                1 << 18
#define CRONO_PCIE_UNC_ECRC_ERROR                   1 << 19
#define CRONO_PCIE_UNC_UNSUPPROTED_REQUEST_ERROR    1 << 20
```

4.5 Configuration

The device is configured with a configuration structure. The user should first obtain a structure that contains the default settings of the device read from an on-board ROM, then modify the structure as needed for the user application and use the result to configure the device.

`int timetagger4_configure(timetagger4_device *device, timetagger4_configuration *config)`
Configures the `timetagger4_manager`.

`int timetagger4_get_current_configuration(timetagger4_device *device, timetagger4_configuration *config)`
Gets current configuration. Copies the current configuration to the specified config pointer.

`int timetagger4_get_default_configuration(timetagger4_device *device, timetagger4_configuration *config)`
Gets default configuration. Copies the default configuration to the specified config pointer.

4.5.1 Structure timetagger4_configuration

This is the structure containing the configuration information. It is used in conjunction with `timetagger4_get_default_configuration()`, `timetagger4_get_current_configuration()` and `timetagger4_configure()`.

It uses multiple substructures to configure various aspects of the board.

`int size`
The number of bytes occupied by the structure.

`int version`
A version number that is increased when the definition of the structure is changed. The increment can be larger than one to match driver version numbers or similar.

`int` **`tdc_mode`**

TDC mode. Can be grouped or continuous defined as follows:

```
#define TIMETAGGER4_TDC_MODE_GROUPED      0
#define TIMETAGGER4_TDC_MODE_CONTINUOUS   1
```

- Grouped functionality is explained in Section 3.1.
- Continuous mode is explained in Section 3.3. The `auto_trigger_period` needs to be set appropriately and `channel[i].stop` must be larger than `auto_trigger_period` (respecting the different periods or can be set to maximum of `0xFFFFFFFF`), if all events need to be captured.

`crono_bool_t` **`start_rising`**

Not applicable for the TimeTagger4. Rising and/or falling edge are configured using the `timetagger4_trigger` structure (see Section 4.5.2).

`double` **`dc_offset[TIMETAGGER4_TDC_CHANNEL_COUNT + 1]`**

Set the threshold voltage for the input channels S, A …D (see Figure 4.1).

- dc_offset[0] : threshold for channel Start
- dc_offset[1 - 4] : threshold for channels A …D

The supported range is -1.27 to 1.13 V. This should be close to 50% of the height of the input pulse. Examples for various signaling standards are defined as follows.

Important Note: The supported range changed for driver release 1.10.7. That means, if you use a value for `dc_offset` outside the new supported range in your source code, the device configuration will adjust it automatically to the new supported range (e.g., a value of +1.18 V will be reduced to +1.13 V).

```
#define TIMETAGGER4_DC_OFFSET_P_NIM         +0.35
#define TIMETAGGER4_DC_OFFSET_P_CMOS        +1.13
#define TIMETAGGER4_DC_OFFSET_P_LVCMOS_33   +1.13
#define TIMETAGGER4_DC_OFFSET_P_LVCMOS_25   +1.13
#define TIMETAGGER4_DC_OFFSET_P_LVCMOS_18   +0.90
#define TIMETAGGER4_DC_OFFSET_P_TTL         +1.13
#define TIMETAGGER4_DC_OFFSET_P_LVTTL_33    +1.13
#define TIMETAGGER4_DC_OFFSET_P_LVTTL_25    +1.13
#define TIMETAGGER4_DC_OFFSET_P_SSTL_3      +1.13
#define TIMETAGGER4_DC_OFFSET_P_SSTL_2      +1.13
#define TIMETAGGER4_DC_OFFSET_N_NIM         -0.35
#define TIMETAGGER4_DC_OFFSET_N_CMOS        -1.27
#define TIMETAGGER4_DC_OFFSET_N_LVCMOS_33   -1.27
#define TIMETAGGER4_DC_OFFSET_N_LVCMOS_25   -1.25
```

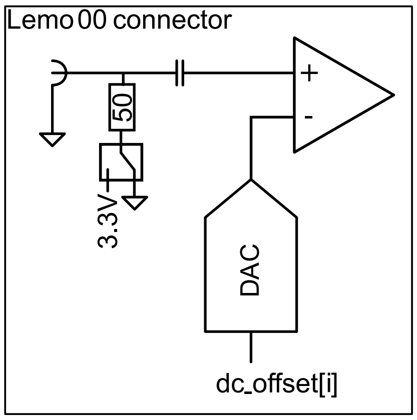


Figure 4.1 Input circuit for each of the input channels.

```
#define TIMETAGGER4_DC_OFFSET_N_LVCMOS_18  -0.90
#define TIMETAGGER4_DC_OFFSET_N_TTL        -1.27
#define TIMETAGGER4_DC_OFFSET_N_LVTTL_33   -1.27
#define TIMETAGGER4_DC_OFFSET_N_LVTTL_25   -1.25
#define TIMETAGGER4_DC_OFFSET_N_SSTL_3     -1.27
#define TIMETAGGER4_DC_OFFSET_N_SSTL_2     -1.25
```

The inputs are AC coupled. Thus, the absolute voltage is not important for pulse inputs. It is the relative pulse amplitude that causes the input circuits to switch. The parameter must be set to the relative switching voltage for the input standard in use. If the pulses are negative, a negative switching threshold must be set and vice versa.

`timetagger4_trigger` **`trigger[TIMETAGGER4_TRIGGER_COUNT]`**
Configuration of the polarity of the external trigger sources (see Section 4.5.2). These are used as inputs for the TiGer blocks and as inputs to the time measurement unit.

`timetagger4_tiger_block` **`tiger_block[TIMETAGGER4_TIGER_COUNT]`**
Configuration of the timing generators (TiGer, see Section 4.5.3).

Index 0 refers to the Start channel; indices 1 through 4 to the Stop channels A through D.

`timetagger4_channel` **`channel[TIMETAGGER4_TDC_CHANNEL_COUNT]`**
Configuration of the TDC channels.

`timetagger4_lowres_channel`
`timetagger4_lowres_channel[TIMETAGGER4_LOWRES_CHANNEL_COUNT]`
Not applicable for the TimeTagger4.

`uint32_t` **`auto_trigger_period`**
`uint32_t` **`auto_trigger_random_exponent`**
Create a trigger either periodically or randomly. There are two parameters

$$M = \texttt{auto\_trigger\_period}$$
$$N = \texttt{random\_exponent}$$

that result in a distance between triggers of T clock cycles.

If the autotrigger is used for the continuous mode the following boundaries apply.

$$T = M + [1...2^N] - 1$$
$$31 \leq M < 78\,125\,000$$
$$0 \leq N < 32$$

Otherwise, the parameters can be used with the following boundaries.

$$T = M + [1...2^N] - 1$$
$$M_{\text{min}} \leq M < 2^{32}$$
$$0 \leq N < 32$$

where M_{min} is 6 for Gen 1 and 8 for Gen 2. There is no enable or reset. The auto-trigger is running continuously. The usage of this trigger can be configured in the TiGer block source field.

`timetagger4_delay_config` **`timetagger4_delay_config[TIMETAGGER4_TDC_CHANNEL_COUNT+1]`**
Configuration of the channel delay values

`uint32_t` **`ignore_empty_packets`**
If enabled (any value but 0), do not write empty packets to the output stream. Disabled by default.

4.5.2 Structure timetagger4_trigger

For each input, this structure determines whether rising or falling edges on the inputs create trigger events for the TiGer blocks.

`crono_bool_t` **`falling`**
`crono_bool_t` **`rising`**
Select for which edges a trigger event is created inside the FPGA. Set the corresponding flag for one of the edges or both edges when using the input with a TiGer.

4.5.3 Structure timetagger4_tiger_block

See Section 3.5 for additional information.

`crono_bool_t` **`enable`**
Activates the timing generator (TiGer).

`crono_bool_t` **`negate`**
Inverts output polarity. Default is set to `false`.

`crono_bool_t` **`retrigger`**
Enables re-trigger setting.

If enabled the timer is reset to the value of the *start* parameter, whenever the input signal is set while waiting to reach the *stop* time.

`crono_bool_t` **`extend`**
Not implemented.

`crono_bool_t` **`enable_lemo_output`**

Enables the LEMO output. Drive the TiGer signal to the corresponding LEMO connector as an output. This is DC coupled, so make sure that you do not connect any devices as inputs. Pulses created by the TiGer are visible at the inputs of the TimeTagger4 and can be measured again to get the exact timing.

`uint32_t` **`start`**
`uint32_t` **`stop`**

In multiples of 4 ns for Gen 1 and 3.2 ns for Gen 2 TimeTagger4. The time during which the TiGer output is set, relative to the trigger input.

The parameters `start` and `stop` must fulfill the following conditions

$$0 \leq \texttt{start} \leq \texttt{stop} \leq 2^{16} - 1\,.$$

If retriggering is enabled, the timer is reset to the value of the start parameter whenever the input signal is set while waiting for the stop time.

`int` **`sources`**

A bit mask with a bit set for all trigger sources that can trigger this TiGer block. Default is `TIMETAGGER4_TRIGGER_SOURCE_S`.

```
#define TIMETAGGER4_TRIGGER_SOURCE_NONE   0x00000000
#define TIMETAGGER4_TRIGGER_SOURCE_S      0x00000001
#define TIMETAGGER4_TRIGGER_SOURCE_A      0x00000002
#define TIMETAGGER4_TRIGGER_SOURCE_B      0x00000004
#define TIMETAGGER4_TRIGGER_SOURCE_C      0x00000008
#define TIMETAGGER4_TRIGGER_SOURCE_D      0x00000010
#define TIMETAGGER4_TRIGGER_SOURCE_AUTO   0x00004000
#define TIMETAGGER4_TRIGGER_SOURCE_ONE    0x00008000
```

4.5.4 Structure timetagger4_channel

Contains TDC channel settings.

`crono_bool_t` **`enabled`**

Enable the TDC channel.

`crono_bool_t` **`rising`**

Not applicable for TimeTagger4. Rising and/or falling edge are configured using the `timetagger4_trigger` structure (see Section 4.5.2).

`uint32_t` **`start`**
`uint32_t` **`stop`**

Veto function for grouping of hits into packets in multiples of the binsize. Only hits between start and stop are read out. The parameters must adhere to the following relations:

$$0 \leq \texttt{start} \leq \texttt{stop} < 2^{31}$$

4.5.5 Structure timetagger4_delay_config

Contains configurable delay value for TimeTagger4 Gen 2 (see Section 3.4).

`uint32_t` **`delay`**

Delay in `static_info.delay_bin_size` (currently 200 ps) for a channel. The possible values are the following

$$0 \leq \texttt{delay} \leq 1023$$

5 Run Time Control

5.1 Run Time Control

Once the devices are configured the following functions can be used to control the behavior of the devices. All of these functions return quickly with very little overhead, but they are not guaranteed to be thread safe.

`int` **`timetagger4_start_capture`**`(timetagger4_device *device)`

Start data acquisition.

`int` **`timetagger4_pause_capture`**`(timetagger4_device *device)`

Pause a started data acquisition.

`pause` and `continue` have less overhead than start and stop but don't allow for a configuration change.

`int` **`timetagger4_continue_capture`**`(timetagger4_device *device)`

Call this to resume data acquisition after a call to `timetagger4_pause_capture()`.

`pause` and `continue` have less overhead than start and stop but don't allow for a configuration change.

`int` **`timetagger4_stop_capture`**`(timetagger4_device *device)`

Stop data acquisition.

`int` **`timetagger4_start_tiger`**`(timetagger4_device *device)`
`int` **`timetagger4_stop_tiger`**`(timetagger4_device *device)`

Start and stop the timing generator. This can be done independently of the state of the data acquisition.

5.2 Readout

The device provides a stream of packets, that are read in batches. A batch of packets is provided to the application, it processes them, by storing important information in other structures. The batch that were processed need to be acknowledged, so that the device can reuse the memory of these for the next data. That means processing should be fast.

```
timetagger4_read_in in;
// automatically acknowledge all data as processed
in.acknowledge_last_read = 1;
volatile crono_packet* p = read_data.first_packet;
timetagger4_read_out out;
int status = timetagger4_read(device, &in, &out);
if (status == CRONO_READ_OK) {
    while (p <= read_data.last_packet) {
        processPacket(p);
        p = crono_next_packet(p);
    }
}
```

`int timetagger4_acknowledge(timetagger4_device *device, crono_packet *packet)`

Acknowledges the processing of the last read block. This is only necessary if `timetagger4_read()` is not called with `in.acknowledge_last_read` set.

This feature allows to either free up partial DMA space early if there will be no call to `timetagger4_read()` anytime soon. It also allows keeping data over multiple calls to `timetagger4_read()` to avoid unnecessary copying of data.

`int timetagger4_read(timetagger4_device *device, timetagger4_read_in *in, timetagger4_read_out *out)`

Return a pointer to an array of captured data in read_out. The result contains a batch of packets of type timetagger4_packet. The batch is described by `first_packet` and `last_packet` in the `timetagger4_read_in` structure.

`read_in` provides parameters to the driver. A call to this method automatically allows the driver to reuse the memory returned by the previous call if `read_in.acknowledge_last_read` is set.

Returns an error code as defined in the structure `timetagger4_read_out`.

`crono_packet crono_next_packet(crono_packet *packet)`

Iterates to the next packet in the batch.

5.2.1 Input Structure timetagger4_read_in

`crono_bool_t acknowledge_last_read`

If set `timetagger4_read()` automatically acknowledges packets from the last read. Otherwise, `timetagger4_acknowledge()` needs to be called explicitly by the user.

5.2.2 Input Structure timetagger4_read_out

`crono_packet *first_packet`

Pointer to the first packet that was captured by the call of `timetagger4_read()`.

`crono_packet *last_packet`

Address of header of the last packet in the buffer. This packet is still valid, all data after this packet is invalid.

`int error_code`

Assignments of the error codes.

```
#define CRONO_READ_OK              0
#define CRONO_READ_NO_DATA         1
#define CRONO_READ_INTERNAL_ERROR  2
#define CRONO_READ_TIMEOUT         3
```

`const char *error_message`

The last error in human-readable form, possibly with additional information about the error.

6 Output Data Format

6.1 Memory Management

The *host buffer* is memory on the host's system in which the data recorded by the TimeTagger4 is stored until it is acknowledged by the user.

The host buffer is managed by the DMA (direct memory access) driver. The DMA driver can only ever write to the host buffer if enough memory is free. That means, new packets will never overwrite old packets, unless they have been acknowledged.

If the host buffer is full, data may be lost. Then, the `CRONO_PACKET_FLAG_HOST_BUFFER_FULL` bit of `crono_packet::flags` is set. To ensure that this does not happen, the user must acknowledge packets fast enough by the analysis software. Note that data only has been lost due to a full host buffer if the `CRONO_PACKET_FLAG_TRIGGER_MISSED` bit of `crono_packet::flags` is set.

6.1.1 Acknowledge Packets

A packet in the host buffer will only be overwritten if it has been acknowledged. This can be done manually by the user by calling `ndigo_acknowledge()` or automatically by the driver if in the call of `ndigo_read()`, `acknowledge_last_read` of the `ndigo_read_in` structure `in` was set to `true` (see Section 5).

Acknowledging a packet acknowledges all previous packets as well.

Be aware that acknowledging a packet *immediately* invalidates it, and it immediately becomes unsafe to attempt accessing its content.

6.1.2 TimeTagger4-Internal Memory Layout

The TimeTagger4 uses internal FIFO (first-in, first-out) memories. In one of these FIFOs, referred to as the DMA FIFO, packets that are ready to be sent to the host system are buffered. If the DMA FIFO is full at any point, the affected packets `CRONO_PACKET_FLAG_DMA_FIFO_FULL` bit of `crono_packet::flags` is set. This does not mean that data has been necessarily lost. Only if the `CRONO_PACKET_FLAG_TRIGGER_MISSED` bit is set has data been lost.

6.2 Output Structure crono_packet

Output of a read call list is a group of `crono_packet` structures. Which have a variable length. The structure contains the following fields.

`uint8_t` **`channel`**

Index of the source channel of the data. Pseudo channel `15` is used for rollovers.

`uint8_t` **`card`**

Identifies the source card in case there are multiple boards present. Defaults to `0` if no value is assigned to the parameter `board_id` in structure `timetagger4_init_parameters`.

uint8_t **type**

The data stream consists of 32-bit unsigned data as signified by `CRONO_PACKET_TYPE_32_BIT_UNSIGNED = 6`.

uint8_t **flags**

Bit field of TIMETAGGER4 _PACKET_FLAG_* bits:

`#define TIMETAGGER4_PACKET_FLAG_ODD_HITS 1`
If this bit is set, the last data word in the data array consists of one timestamp only which is located in the lower 32 bits of the 64-bit data word (little endian).

`#define TIMETAGGER4_PACKET_FLAG_SLOW_SYNC 2`
Timestamp of a hit is above the range of 8-bit rollover number and 24-bit hit timestamp. The group is closed, all other hits are ignored.

`#define TIMETAGGER4_PACKET_FLAG_START_MISSED 4`
The trigger unit has discarded packets due to a full FIFO because the data rate is too high. Starts are missed and stops are potentially in wrong groups.

`#define TIMETAGGER4_PACKET_FLAG_SHORTENED 8`
The trigger unit has shortened the current packet due to a full pipeline FIFO because the data rate is too high. Stops are missing in the current packet.

`#define TIMETAGGER4_PACKET_FLAG_DMA_FIFO_FULL 16`
The internal DMA FIFO was full. This is caused either because the data rate is too high on too many channels. Packet loss is possible.

`#define TIMETAGGER4_PACKET_FLAG_HOST_BUFFER_FULL 32`
The host buffer was full. Might result in dropped packets. This is caused either because the data rate is too high or by data not being retrieved fast enough from the buffer. Solutions are increasing buffer size if the overload is temporary or by avoiding or optimizing any additional processing in the code that reads the data.

uint32_t **length**

Number of 64-bit elements (each containing up to 2 TDC hits) in the data array. The number of hits contained is equal to `2 * length - (flags & PACKET_FLAG_ODD_HITS) ? 1 : 0`.

uint64_t **timestamp**

Coarse timestamp of the start pulse. Values are given in multiples of `packet_binsize` contained in `timetagger4_param_info`.

uint64_t **data[1]**

Contains the TDC hits as a variable length array (length can be zero). The user can cast the array to `uint32_t*` to directly operate on the TDC hits. For the number of hits, see length. Structure of one hit (32 bit):

bits	31 to 8	7 to 4	3 to 0
content	TDC DATA	FLAGS	CHN

The timestamp of the hit is stored in bits 31 down to 8 in multiples of `binsize` contained in `timetagger4_param_info`.

```
uint32_t timestamp  = (hit >> 8) & 0xF;
uint32_t flags      = (hit >> 4) & 0xF;
uint32_t channel    =  hit       & 0xF;
```

Bits 7 down to 4 are hit flags and have the following definitions:

- Bit 7: Not applicable for the TimeTagger4 and therefore always 0.
- `#define TIMETAGGER4_HIT_FLAG_COARSE_TIMESTAMP` 4 ↔ Bit 6
  Bit 6: Always 1 for the TimeTagger4.
- `#define TIMETAGGER4_HIT_FLAG_TIME_OVERFLOW` 2 ↔ Bit 5
  Bit 5: If set, this hit is a rollover. The time since the start pulse exceeded the 24-bit range that can be encoded in a data word. This word does not encode a measurement. Instead, the readout software should increment a rollover counter that can be used as the upper bits of consecutive time stamps. The counters must be reset for each packet. The total offset of a hit in picoseconds can be computed by

  $$\Delta T_{hit} = (\#\text{rollovers} \times \texttt{timetagger4\_static\_info.rollover\_period} + \text{TDC\_DATA}_{\text{hit}}) \\ \times \texttt{timetagger4\_param\_info.binsize}$$

- `#define TIMETAGGER4_HIT_FLAG_RISING` 1 ↔ Bit 4
  Bit 4: Set if this hit is a rising edge. Otherwise, this word belongs to a falling edge.

Bits 3 down to 0: The channel number is given in the lowest nibble of the data word. A value of 0 corresponds to channel A, a value of 3 to channel D.

7 Code Example

The following C++ source code shows how to initialize a TimeTagger4 board, configure it and loop over incoming packets.

If you are reading this documentation in portable document format (PDF), the source code of the C example is also embedded as an attachment to the file. You can open it in an external viewer or save it to disk by clicking on it. The source code can also be found on https://github.com/cronologic-de/xtdc_babel/tree/main/timetagger4_user_guide_example.

```
// timetagger4_user_guide_example.cpp : Example application for the ↩
   TimeTagger4
#include <stdio.h>
#include <stdlib.h>
#include <chrono>
#include <thread>
#include "TimeTagger4_interface.h"

// If true the time tagger triggers a start periodically
// The time difference of signals on channel A are measured
// else start signal either from input or tiger is used (see below)
// frequency of  start signal is printed and the hits are sampled
const bool USE_CONTINUOUS_MODE = false;
const bool USE_TIGER_START = true; // if false, external signal must be
                                    // provided on start; not applicable if
                                    // continuous mode is enabled
const bool USE_TIGER_STOPS = true;  // if false please connect signals to
                                    // some of channels A-D


timetagger4_device* initialize_timetagger(int buffer_size,
                                          int board_id,
                                          int card_index){
    // prepare initialization
    timetagger4_init_parameters params;

    timetagger4_get_default_init_parameters(&params);
    params.buffer_size[0] = buffer_size; // size of the packet buffer
    params.board_id = board_id;          // value copied to "card" field of
                                         // every packet, allowed range ↩
                                            0..255
    params.card_index = card_index;      // which of the TimeTagger4 board
                                         // found in the system to be used
    int error_code;
    const char * err_message;
    timetagger4_device* device = timetagger4_init(&params,
                                                  &error_code,
                                                  &err_message);
    if (error_code != CRONO_OK) {
        printf("Could not init TimeTagger4 compatible board: %s\n",
               err_message);
        return nullptr;
    }
    timetagger4_static_info static_info;
    timetagger4_get_static_info(device, &static_info);
    bool timeTaggerNG = static_info.board_revision >= 7;
    if (USE_CONTINUOUS_MODE && !timeTaggerNG) {
        printf("Cannot use continuous mode with TimeTagger 1G/2G: %s\n",
               err_message);
        timetagger4_close(device);
        return nullptr;
    }
    return device;
}

int configure_timetagger(timetagger4_device* device) {
    // prepare configuration
    timetagger4_static_info static_info;
    timetagger4_get_static_info(device, &static_info);
    timetagger4_configuration config;
    // fill configuration data structure with default values
    // so that the configuration is valid and only parameters
    // of interest have to be set explicitly
    timetagger4_get_default_configuration(device, &config);

    // set config of the 4 TDC channels
    for (int i = 0; i < TIMETAGGER4_TDC_CHANNEL_COUNT; i++)
    {
        // enable recording hits on TDC channel
        config.channel[i].enabled = true;

        // define range of the group
        config.channel[i].start = 0; // range begins right after start pulse
        if (!USE_CONTINUOUS_MODE) {
            config.channel[i].stop = 30000; // recording window stops
                                            // after ~15 us
        }
        else {
            config.channel[i].stop = 0x7fffffff; // trigger is independent
                                                 // of stops
                                                 // set to maximal value
        }

        // measure only rising edge for tiger (positive) pulse or falling
        // for user (negative) pulse
        config.trigger[TIMETAGGER4_TRIGGER_A + i].falling =
            USE_TIGER_STOPS ? 0 : 1;
        config.trigger[TIMETAGGER4_TRIGGER_A + i].rising =
            USE_TIGER_STOPS ? 1 : 0;
    }

    // generate an internal 25 kHz trigger, used for tiger and continuous ↩
       mode
    config.auto_trigger_period =
        (int)(static_info.auto_trigger_ref_clock / 25000);
    config.auto_trigger_random_exponent = 0;

    // setup TiGeR
    // sending a signal to the LEMO outputs (and to the TDC on the same ↩
       channel)
    // requires proper 50 Ohm termination on the LEMO output to work ↩
       reliably

    // width of the 12ns pulse in the auto_trigger clock periods
    int pulse_width = (int) (12e-9 * static_info.auto_trigger_ref_clock);

    if (!USE_CONTINUOUS_MODE) {
        // use 200 kHz auto trigger to generate

        // generate above configured auto trigger to generate a
        // signal with 12 ns pulse width on LEMO output Start
        config.tiger_block[0].enable = USE_TIGER_START ? 1 : 0;
        config.tiger_block[0].start = 0;
        config.tiger_block[0].stop = config.tiger_block[0].start + ↩
           pulse_width;
        config.tiger_block[0].negate = 0;
        config.tiger_block[0].retrigger = 0;
        config.tiger_block[0].extend = 0;
        config.tiger_block[0].enable_lemo_output = 1;
        config.tiger_block[0].sources = TIMETAGGER4_TRIGGER_SOURCE_AUTO;
        // if TiGeR is used for triggering with positive pulses
        if (USE_TIGER_START)
            config.dc_offset[0] = TIMETAGGER4_DC_OFFSET_P_LVCMOS_18;
        else // user input expect NIM signal
            config.dc_offset[0] = TIMETAGGER4_DC_OFFSET_N_NIM;

        // start group on falling edges on the start channel 0
        config.trigger[TIMETAGGER4_TRIGGER_S].falling = USE_TIGER_START ? 0 ↩
           : 1;
        config.trigger[TIMETAGGER4_TRIGGER_S].rising = USE_TIGER_START ? 1 :↩
            0;
    } else {
        // Auto trigger is used as a start signal
        config.tdc_mode = TIMETAGGER4_TDC_MODE_CONTINUOUS;
    }

    for (int i = 1; i < TDC4_TIGER_COUNT; i++) {
        config.tiger_block[i].enable = USE_TIGER_STOPS ? 1 : 0;
        config.tiger_block[i].start = i * 100;
        config.tiger_block[i].stop = config.tiger_block[i].start + ↩
           pulse_width;
        config.tiger_block[i].negate = 0;
        config.tiger_block[i].retrigger = 0;
        config.tiger_block[i].extend = 0;
        config.tiger_block[i].enable_lemo_output = USE_TIGER_STOPS ? 1 : 0;
        config.tiger_block[i].sources = TIMETAGGER4_TRIGGER_SOURCE_AUTO;

        if (USE_TIGER_STOPS)
            config.dc_offset[i] = TIMETAGGER4_DC_OFFSET_P_LVCMOS_18;
        else // user input expect NIM signal
            config.dc_offset[i] = TIMETAGGER4_DC_OFFSET_N_NIM;

```

```c
        // this is not related to the tigers, but uses the same indexing (0 is start)
        // optionally increase input delay by 10 * 200 ps for each channel on new TT
        // config.delay_config[i].delay = i * 10;
    }

    // write configuration to board
    return timetagger4_configure(device, &config);
}

void print_device_information(timetagger4_device* device,
                              timetagger4_static_info* si,
                              timetagger4_param_info* pi) {
    // print board information
    printf("Board Serial        : %d.%d\n",
           si->board_serial >> 24, si->board_serial & 0xffffff);
    printf("Board Configuration : %s\n",
           timetagger4_get_device_name(device));
    printf("Board Revision      : %d\n",
           si->board_revision);
    printf("Firmware Revision   : %d.%d\n",
           si->firmware_revision, si->subversion_revision);
    printf("Driver Revision     : %d.%d.%d\n",
           ((si->driver_revision >> 16) & 255),
           ((si->driver_revision >> 8) & 255),
           (si->driver_revision & 255));
    printf("Driver SVN Revision : %d\n",
           si->driver_build_revision);
    printf("\nTDC binsize         : %0.2f ps\n",
           pi->binsize);
}

double last_abs_ts_on_a = 0;
int64_t last_group_abs_time = 0;

int64_t processPacket(volatile crono_packet* p,
                      bool print,
                      timetagger4_static_info* si,
                      timetagger4_param_info* pi){
    // do something with the data, e.g. calculate current rate
    int64_t group_abs_time = p->timestamp;
    if (!USE_CONTINUOUS_MODE) {
        // group timestamp increments at binsize, but we see only a fraction of
        // the packets (every update_count)
        double rate = 1e12 / (
                (double)(group_abs_time - last_group_abs_time)
                * pi->packet_binsize
            );
        if (print && last_group_abs_time > 0) {
            printf("\r%.6f kHz", rate / 1000.0);
            // ...or print hits (not a good idea at high data rates,
            printf("Card %d - flags %d - length %d - type %d - TS %llu\n",
                   p->card, p->flags, p->length, p->type, p->timestamp);
        }
        last_group_abs_time = group_abs_time;
    }

    int hit_count = 2 * (p->length);
    // Two hits fit into every 64 bit word. The second in the last word might
    // be empty
    // This flag tells us, whether the number of hits in the packet is odd
    if ((p->flags & TIMETAGGER4_PACKET_FLAG_ODD_HITS) != 0)
        hit_count -= 1;

    uint32_t* packet_data = (uint32_t*)(p->data);
    uint32_t rollover_count = 0;
    uint64_t rollover_period_bins = si->rollover_period;
    for (int i = 0; i < hit_count; i++)
    {
        uint32_t hit = packet_data[i];
        uint32_t channel = hit & 0xf;
        // extract hit flags
        uint32_t flags = hit >> 4 & 0xf;


        if ((flags & TIMETAGGER4_HIT_FLAG_TIME_OVERFLOW) != 0) {
            // this is a overflow of the 23/24 bit counter)
            rollover_count++;
        }
        else {
            // extract channel number (A-D)
            char channel_letter = 65 + channel;

            // extract hit timestamp
            uint32_t ts_offset = hit >> 8 & 0xffffff;

            // Convert timestamp to ns, this is relative to the start of
            // the group
            double ts_offset_ns =
                (ts_offset + rollover_count * rollover_period_bins)
                * pi->binsize / 1000.0;

            if (USE_CONTINUOUS_MODE) {
                if (channel == 0) {
                    // compute the absolute time by adding the group time in ns
                    double abs_ts_on_a =
                        (group_abs_time * pi->packet_binsize) / 1000
                        + ts_offset_ns;
                    double diff = abs_ts_on_a - last_abs_ts_on_a;
                    if (last_abs_ts_on_a > 0 && print) {
                        printf("Time difference between hits on A  %.1f ns\n",
                               diff);
                    }
                    last_abs_ts_on_a = abs_ts_on_a;
                }
            }
            else {
                if (print)
                    printf("Hit  on channel %c - flags %d - offset %u (raw) ↩
                       / %.1f ns\n",
                           channel_letter, flags, ts_offset, ts_offset_ns);
            }
        }
    }
    return group_abs_time;
}

int main(int argc, char* argv[]) {
    printf("cronologic timetagger4_user_guide_example using driver: %s\n",
           timetagger4_get_driver_revision_str());
    timetagger4_device* device = initialize_timetagger(8 * 1024 * 1024, 0, ↩
       0);
    if (device == nullptr) {
        exit(1);
    }
    int status = configure_timetagger(device);
    if (status != CRONO_OK) {
        printf("Could not configure TimeTagger4: %s",
               timetagger4_get_last_error_message(device));
        timetagger4_close(device);
        return status;
    }
    timetagger4_static_info static_info;
    timetagger4_get_static_info(device, &static_info);

    timetagger4_param_info parinfo;
    timetagger4_get_param_info(device, &parinfo);

    print_device_information(device, &static_info, &parinfo);

    // configure readout behaviour
    timetagger4_read_in read_config;
    // automatically acknowledge all data as processed
    // on the next call to timetagger4_read()
    // old packet pointers are invalid after calling timetagger4_read()
    read_config.acknowledge_last_read = 1;

    // structure with packet pointers for read data
    timetagger4_read_out read_data;

    // start data capture
    status = timetagger4_start_capture(device);
    if (status != CRONO_OK) {
        printf("Could not start capturing %s",
               timetagger4_get_last_error_message(device));
        timetagger4_close(device);
        return status;
    }

    // start timing generator
    timetagger4_start_tiger(device);

    // some book keeping
    int packet_count = 0;
    int empty_packets = 0;
    int packets_with_errors = 0;
    bool last_read_no_data = false;

    int64_t group_abs_time = 0;
    int64_t group_abs_time_old = 0;
    int update_count = 100;

    printf("Reading packets:\n");
    bool no_data_printed = false;
    // read 10000 packets
    while (packet_count < 10000)
    {
        // get pointers to acquired packets
        status = timetagger4_read(device, &read_config, &read_data);
        if (status != CRONO_OK) {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            // to avoid a lot of lines with no data
            if (!no_data_printed) {
                printf(" - No data! -\n");
                no_data_printed = true;
            }
        }
        else
        {
            // iterate over all packets received with the last read
            volatile crono_packet* p = read_data.first_packet;
            while (p <= read_data.last_packet)
            {
                // printf is slow, so this demo only processes every nth packet
                // your application would of course collect every packet
                bool print = packet_count % update_count == 0;

                processPacket( p, print, &static_info, &parinfo);
                no_data_printed = false;
                p = crono_next_packet(p);
                packet_count++;
            }
        }
    }

    // shut down packet generation and DMA transfers
    timetagger4_stop_capture(device);

    // deactivate timetagger4
    timetagger4_close(device);
    return 0;
}
```

8 Technical Data

Each board is tested against the values listed in the columns “Min” and “Max”. “Typical” is the mean value of the first 10 boards produced or a value that is set by design.

8.1 TDC Characteristics

8.1.1 TDC measurement Characteristics for Gen 1 TimeTagger4

Symbol	Parameter	Min	Typical	Max	Units
INL	Integral non-linearity			0.5	bins
DNL	Differential non-linearity		0.2		bins
t_{Data}	Data format bin size		500		ps
t_{Res1}	Double pulse resolution for -1G		1000		ps
t_{Res2}	Double pulse resolution for -2G		500		ps
Δt_{Start}	Interval between consecutive starts	4			ns
t_{Range}	Measurement range using hits only			8.368	ms
$t_{Extended}$	Extended range using rollovers			2.147	s
$f_{Readout}$	Readout rate			48	MHits/s

8.1.2 TDC measurement Characteristics for Gen 2 TimeTagger4

Symbol	Parameter	Min	Typical	Max	Units
INL	Integral non-linearity			0.5	bins
DNL	Differential non-linearity −1.25G to −5G −10G			 0.01 0.3	t_{Data}
t_{Data}	Data format bin size		100		ps
t_{Quant}	Quantization −1.25G −2.5G −5G −10G		 800 400 200 100		ps
t_{Res}	Double pulse resolution		2		t_{Quant}
t_{Range}	Measurement range using hits only			1.677	ms
$t_{Extended}$	Extended range using rollovers			0.429	s
$f_{Start,burst}$	Burst rate for up to 4096 starts −1.25G −2.5G −5G −10G			 0.625 1.25 2.5 5	GHz
$f_{Start,sust}$	Sustained rate of starts			18	MHz
$f_{Stop,burst}$	Burst rate for up to 3900 stops −1.25G −2.5G −5G −10G			 0.625 1.25 2.5 5	GHits/s
$f_{Readout,single}$	Readout rate per channel			40	MHits/s
$f_{Readout,all}$	Readout rate of all channels			60	MHits/s

8.1.3 Oscillator Time Base

Symbol	Parameter	Min	Typical	Max	Units
ΔT	Temperature stability 20 °C to 70 °C			250	ppb
F	Initial calibration			1	ppm
$\Delta F/F_1$	Aging first year			2	ppm
$\Delta F/F_{10}$	Aging 10 years			8	ppm

8.2 Electrical Characteristics

8.2.1 Power Supply

Symbol	Parameter	Min	Typical	Max	Units
P_{total}	Total power consumption			10	W
$VCC_{3.3}$	PCIe 3.3 V rail power supply voltage	3.1	3.3	3.5	V
$I_{3.3}$	PCIe 3.3 V rail input current			650	mA
VCC_{12}	PCIe 12 V rail power supply voltage	11.1	12.0	12.9	V
I_{12}	PCIe 12 V rail input current			550	mA
VCC_{aux}	PCIe 3.3 V_{Aux} rail power supply voltage		3.3		V
I_{aux}	PCIe 3.3 V_{Aux} rail input current		0		mA

8.2.2 TDC Inputs

The TimeTagger4's inputs are single-ended AC-coupled with 50 Ω termination.

Symbol	Parameter	Min	Typical	Max	Units
V_{Base}	Input Baseline	0		5	V
$V_{Threshold}$	Trigger Level	$V_{Base} - 1.27$		$V_{Base} + 1.13$	V
t_{Pulse}	Pulse Length		5	200	ns
	–1.25G	0.8			
	–2.5G	0.4			
	–5G	0.2			
	–10G	0.1			
t_{Rise}	Pulse Edge 20% to 80%			10	ns
t_{Fall}	Pulse Edge 80% to 20%			10	ns
Z_P	Input Impedance		50		Ω
I_{Term}	Termination Current	–50	–20	50	mA

All inputs are AC-coupled. The inputs have very high input bandwidth requirements and therefore there are no circuits that provide over-voltage protection for these signals. Any voltage on the inputs above 5 V or below −5 V relative to the voltage of the slot cover can result in permanent damage to the board.

Keep in mind, that the input baseline V_{Base} is affected by the ratio of pulse length t_{Pulse} to average pulse distance (for continuous signals the term is called duty cycle).

Make sure not to drive the inputs when the connector is configured as a TiGer output.

See Section 3.5.

8.3 Information Required by DIN EN 61010-1

8.3.1 Manufacturer

The TimeTagger4 is a product of:

cronologic GmbH & Co. KG
Jahnstraße 49
60318 Frankfurt

Germany HRA 42869 beim Amtsgericht Frankfurt/M

VAT-ID: DE235184378
PCI Vendor ID: 0x1A13

8.3.2 Intended Use and System Integration

The devices are not ready to use as delivered by cronologic. It requires the development of specialized software to fulfill the application of the end-user. The device is provided to system integrators to be built into measurement systems that are distributed to end users. These systems usually consist of the TimeTagger4, a main board, a case, application software and possibly additional electronics to attach the system to some type of detector. They might also be integrated with the detector.

The TimeTagger4 is designed to comply with DIN EN 61326-1 when operated on a PCIe compliant main board housed in a properly shielded enclosure. When operated in a closed standard compliant enclosure the device does not pose any hazards as defined by EN 61010-1.

Radiated emissions, noise immunity, and safety highly depend on the quality of the enclosure. It is the responsibility of the system integrator to ensure that the assembled system is compliant to applicable standards of the country that the system is operated in, especially regarding user safety and electromagnetic interference.

When handling the board, adequate measures must be taken to protect the circuits against electrostatic discharge (ESD). All power supplied to the system must be turned off before installing the board.

8.3.3 Environmental Conditions for Storage

The board shall be stored between operation under the following conditions:

Symbol	Parameter	Min	Typical	Max	Units
T_{store}	ambient temperature	–30		60	°C
RH_{store}	relative humidity at 31°C noncondensing	10		70	%

8.3.4 Environmental Conditions for Operation

The board is designed to be operated under the following conditions:

Symbol	Parameter	Min	Typical	Max	Units
T_{oper}	ambient temperature	5		40	°C
RH_{oper}	relative humidity at 31°C	20		75	%

WARNING: Do not connect any DC-coupled inputs to a channel while the TiGer of that channel is configured as an output (see Section 3.5). Doing so could permanently damage the TimeTagger4 and the external hardware.

8.3.5 Cooling

The TimeTagger4 in its base configuration has passive cooling that requires a certain amount of air-flow. If the case design can't provide enough air-flow to the board, a slot cooler like Zalman ZM-SC100 can be placed next to the board. Active cooling is also available as an option for the board.

8.3.6 Recycling

cronologic is registered with the "Stiftung Elektro-Altgeräte Register" as a manufacturer of electronic systems with Registration ID DE 77895909.

The TimeTagger4 belongs to category 6, "Kleine Geräte der Informations- und Telekommunikationstechnik für die ausschließliche Nutzung in anderen als privaten Haushalten." Devices sold before 2018 belong to category 9, "Überwachungs und Kontrollinstrumente für ausschließlich gewerbliche Nutzung." The last owner of a TimeTagger4 must recycle it, treat the board in compliance with §11 and §12 of the German ElektroG, or return it to the manufacturer's address listed on Page 43.

9 Ordering Information

TimeTagger4 – 1.25G – TBT

1.25G	1.25 Gsps / 800 ps binsize
2.5G	2.5 Gsps / 400 ps binsize
5G	5 Gsps / 200 ps binsize
10G	10 Gsps / 100 ps binsize

TBT	External device connection to Thunderbolt ports
PCIe	PCIe CEM x1 plugin board

10 Revision History

User Guide 1.10.4 as of 2024-09-19
cronologic GmbH & Co. KG
Jahnstraße 49
60318 Frankfurt am Main
Germany

10.1 Firmware Gen 1

Revision	Date	Comments
0.1219	2024-07-15	PCIe interface optimizations Fixed first-level FIFO overflow Fixed rare bug that causes PC freezes
0.1187	2023-05-25	Extended standard range of measurement to 24 bits Fixed wrong polarity flag bug Internal optimizations
0.1132	2021-12-09	Fixed possible register read issues
0.1118	2021-06-23	Fixed register write issues
0.983	2019-03-15	Internal optimizations
0.971	2019-02-19	Hit sorting and packet generation issues fixed

10.2 Firmware Gen 2

Revision	Date	Comments
0.24178	2024-06-26	Fixed bug related to the PCIe interface Internal optimizations
0.24052	2024-02-21	Activated AER capabilities Improved alignment for TT4-10G Added jitter filter for TT4-10G Added option to discard empty packets before they reach the host buffer
0.23180	2023-06-29	Initial release

10.3 Driver & Applications

Revision	Date	Comments
1.10.7	2024-07-25	Support for TimeTagger4-10G (incl. calibration tool) Include baseline calibration Reduced supported range of `dc_offset` by 100 mV.
1.9.0	2023-07-10	Added quantization to timetagger4_param_info structure Code refactorization
1.8.3	2023-06-07	Minor bug fixes Code refactorization
1.8.2	2023-05-17	Added bounds and checks for various parameters
1.8.1	2023-05-09	Renamed autotrigger mode to continuous mode
1.8.0	2023-05-05	Added configurable input delay
1.7.0	2023-04-18	Board Revision 7 support TimeTagger4 : added autotrigger mode
1.4.5	2022-10-17	kernel driver v1.4.2 for xTDC4 only (fixes crash on Windows for Thunderbolt hot-plug)
1.4.4	2022-06-27	kernel driver v1.4.1
1.4.2	2021-07-28	Firmware updated ReadoutGUI added/updated User guide example added/updated
1.4.1	2019-11-11	x64 32 mode issues fixed
1.4.0	2019-06-04	Improved Windows 10 support
1.3.0	2019-01-23	Added Windows 10 support

10.4 User Guide

Revision	Date	Comments
1.10.4	2025-03-20	TimeTagger4: Updated Table 8.2.2
1.10.3	2025-03-18	xHPTDC8: Documented Firmware 2.1225 and Driver 1.5.0
1.10.2	2024-12-17	xHPTDC8: Updated Figure 2.3 xTDC4: Fixed formatting
1.10.1	2024-09-19	xHPTDC8: Fixes and additions to introduction xHPTDC8 and TimeTagger4: Documented max. readout rate for single channels Updated Figure 2.4
1.10.0	2024-08-14	TimeTagger4: Renamed 10G calibration tool Added Section “Memory Layout”
1.9.4	2024-07-30	Updated driver and firmware revision lists xHPTDC8: Updated user guide `example.cpp`

continued from previous page

Revision	Date	Comments
1.9.3	2024-07-16	xHPTDC8: Fix driver revision list
1.9.2	2024-07-09	xHPTDC8: Added LED documentation TimeTagger4 and xTDC4: Add overview figure of TBT and PCIe variant Fixed grammar
1.9.1	2024-07-02	xHPTDC8: Updated firmware list
1.9.0	2024-06-27	Added new driver revision TimeTagger4 and xTDC4: Added TBT variant TimeTagger4 and xTDC4: Added ordering information TimeTagger4 and xTDC4: Updated supported range for `dc_offset`
1.8.17	2024-06-20	xTDC4: Fixed API documentation
1.8.16	2024-06-20	TimeTagger4: Added documentation for 10G calibration tool xTDC4 and TimeTagger4: Added LED documentation xHPTDC8: Fixed default values for zero_channel Clarifications for TiGer block indices
1.8.15	2024-05-08	Fixed `auto_trigger` formula Updated oscillator characteristics xHPTDC8: Fixed mistakes in API xHPTDC8: Updated Code Examples
1.8.14	2024-03-27	Updated API Updated information on power consumption xHPTDC8: Extended chapter on gating
1.8.13	2024-01-18	xHPTDC8: Updated cover TimeTagger4: Updated feature list
1.8.12	2024-01-10	xHPTDC8: Updated driver revision history
1.8.11	2023-11-29	Reformatting Added latency between signal and Tiger output to Section 3.5 TimeTagger4: Updated table in Section 8.1.2 TimeTagger4: Clarifications in Features-list TimeTagger4: Added `ignore_empty_packets` API documentation xHPTDC8: Added default values for manager and configuration structs xHPTDC8: Fixed number of boards that can be synchronized from 8 to 6
1.8.10	2023-07-28	Changed extended range values to 0.429 s and 2.147 s, respectively. API clarifications.
1.8.9	2023-07-10	TimeTagger4 User Guide rework
1.8.8	2023-03-15	New TimeTagger4 variants -1.25G to -10G added
1.8.7	2022-11-24	Firmware revision notes updated
1.8.6	2022-11-23	Spelling and grammar corrections New example source code for xHPTDC8
1.8.5	2021-12-17	Clarifications related to TimeTagger4 configuration.
1.8.4	2021-12-08	Updated grouping structure in xHPTDC8 API
1.8.3	2021-07-28	Updated firmware revision history

continued from previous page

Revision	Date	Comments
1.8.2	2021-04-23	Added software trigger and _SYNC trigger sources for xHPTDC8 Corrected 3.3V power requirement for xHPTDC8 Changed types with fixed bit width to stdint.h for xHPTDC8 Added user flash functions for xHPTDC8
1.8.1	2021-04-09	Many corrections and updates to the xHPTDC8 API
1.8.0	2021-03-22	Added xHPTDC8 User Guide
1.7.0	2021-02-04	Combined User Guide for -1G and -2G Added characteristics for INL, DNL and Time Base Reordered sections for clarity Error corrections for rollovers, binsize and range Added figure 3.2 (TiGer matrix) Corrected board revision
1.3.0	2019-06-05	API clarifications

Erratum

We found undesired behavior for Gen 1 devices of the TimeTagger4.

If there are three or more edges close together (within 6.6 ns) and the user did only enable rising or falling edges but not both, some edges are reported with the wrong polarity.

If your configuration enables both edges all output data is correct. If you only need one type of edge (rising or falling) there are three simple workarounds:

a) update the Firmware of your Gen 1 device to svn`1187` or later.

b) enable both edges.

   All output words will be correct and your software can ignore all data that doesn't have the desired polarity.

c) enable only the desired edge polarity

   Ignore the polarity flag in the output data. You can trust that only edges with the desired polarity are output, even if the flag in the data word states the wrong polarity.