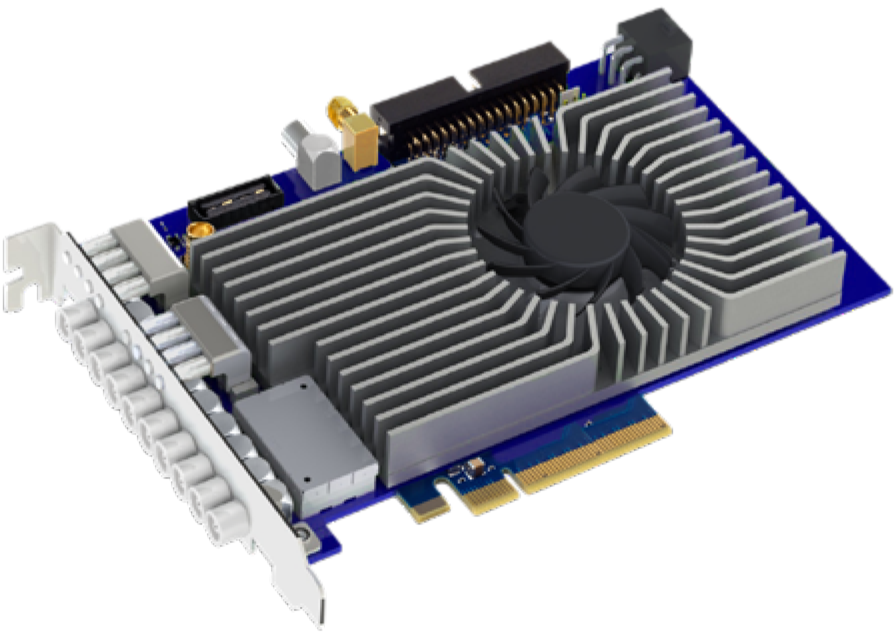
cronologic

Ndigo6G-12

User Guide



www.cronologic.de

Ndigo6G-12

# Contents

# Introduction

The Ndigo6G-12 offers **6400 Msps sample rate**, **12 bit resolution** and a greatly improved **readout rate of up to 5200 MB/s**.

The unit is a **combined ADC/TDC board** for the acquisition of pulses in time-of-flight applications. It builds on the established platform of the Ndigo5G-10 but takes it to the next level both in performance and flexibility.

The Ndigo6G-12 was specifically designed for time-of-flight applications like LIDAR or TOF mass spectrometry. A measurement precision of **5 ps (RMS)** is achievable for unipolar pulses. In addition, information on the pulse shape, such as area or amplitude, is recorded.

**Four channels** with **1600 Msps at 12 bit resolution** can be acquired independently. Alternatively, the four channels can be combined into two channels or into a single channel. This way, either a higher temporal resolution **up to 6400 Msps** or a **larger dynamic range** can be achieved via *multiple-sampling modes*.

This User Guide documents the hardware and functionality of the Ndigo6G-12 board, as well as the driver programming API provided by the Ndigo6G-12 driver.

This User Guide is also available online at docs.cronologic.de/ndigo6g.

## Features

- **12 bit** dynamic range
- Up to **6400 Msps** sample rate (in 1-channel mode) for increased resolution in time domain.
- Up to **four ADC channels** for your individual measurement setups.
- **Four TDC channels** with a resolution of **13 ps**.
- **Two digital control inputs** for effective **gating and triggering**.
- PCIe3 x8 interface for simple and fast data transfer to most PCs.
- **Unlimited multihit** capabilities.
- **Common start** and **common stop** capabilities.
- Continuous ADC readout rate of approx. **5200 MB/s**.
- **Zero suppression**, significantly reducing PCIe load.
- **Internal 10 MHz** clock with a time base of **10 ppb** or the **ability to use an external** 10 MHz clock.

## Board overview

| | |
|---|---|
| Optimized for | TOF applications |
| ADC channels | 4 |
| TDC channels | 4 |
| Digital control channels | 2 |
| Connectors | 10 × LEMO 00 |
| Sample rate | 6400 Msps (1-Channel Mode)<br>3200 Msps (2-Channel Mode)<br>1600 Msps (4-Channel Mode) |
| Resolution | 12 bit |
| Maximum bandwidth | TBD |
| TDC bin size | 13 ps |
| TDC double pulse resolution | typically 5 ns |
| Multihit | unlimited |
| Dead time between groups | none |
| Readout rate | 5200 MByte/s (ADC)<br>30 MHits/s (all TDC channels)<br>11.6 MHits/s (single TDC channel) |
| Timestamp range | 106 d |
| Readout interface | PCIe3 x8 |
| Time base | 10 ppb (internal) or external 10 MHz clock |
| On-board calibration data storage | yes |
| Adjustable trigger windows | yes |
| Possibility for overlapping events | yes |
| Easy-to-use Windows C-API | yes |
| In-system firmware updates | yes |

# 1 Hardware

## 1.1 Installation

The Ndigo6G-12 board can be installed in any PCIe x8 (or higher amount of lanes) PCIe slot. If the slot electrically supports less than eight lanes, the board will operate at lower data throughput rates.

Connect a 6-pin PCIe power cable to the connector at the rear of the board (see Figure 1.1).

**Note**

The Ndigo6G-12 does not operate without a 6-pin PCIe power connector.

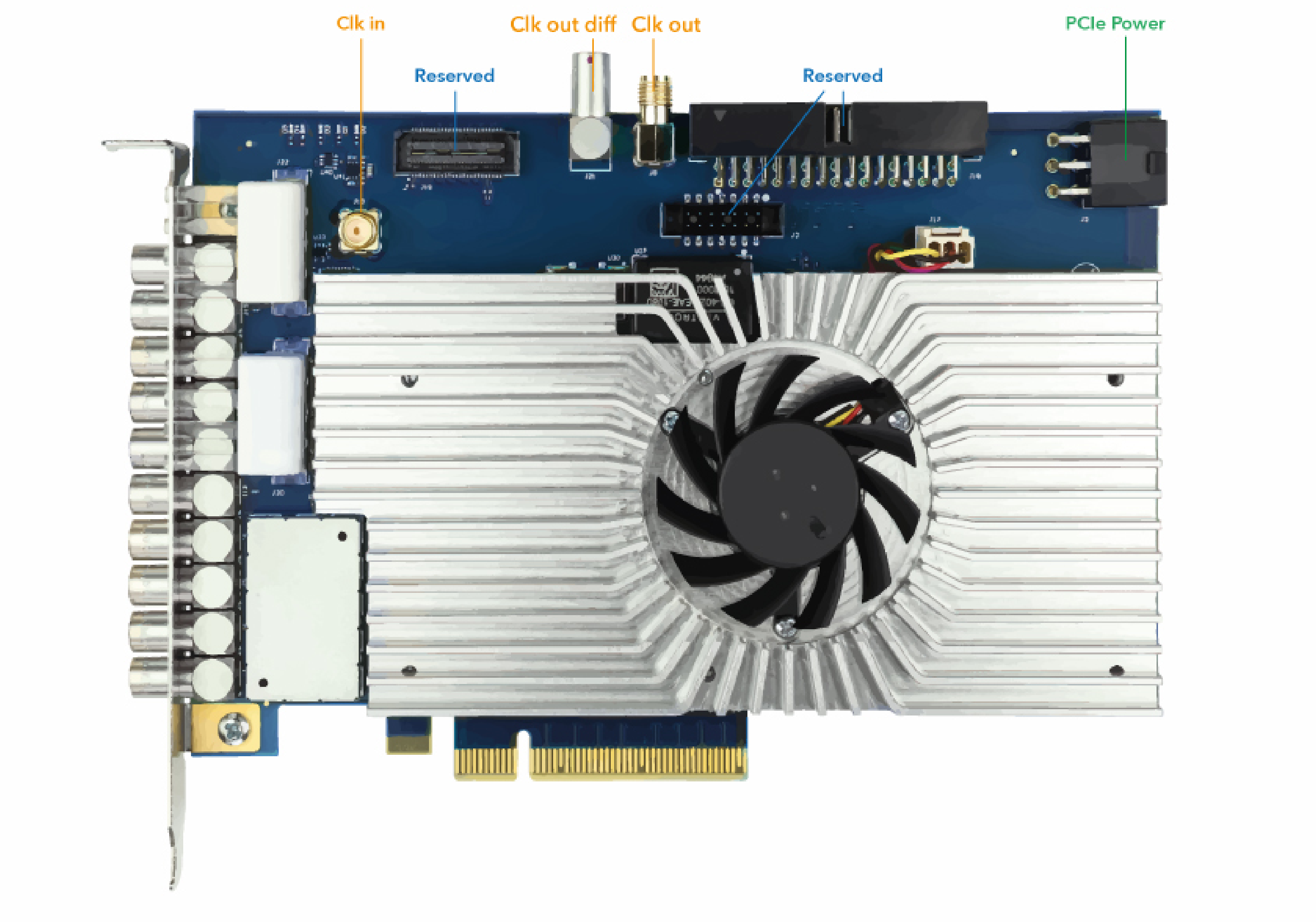



**Figure 1.1:** Overview of an Ndigo6G-12 board. Note the PCIe power connector at the rear of the board.

## 1.2 Cooling

The Ndigo6G-12 board is equipped with an active cooling system, ensuring proper cooling of the device. If, however, the temperature of the ADC chip exceeds 90 °C (for instance, if the device is operated in inappropriate environmental conditions, see Section 6.3.1), a warning is issued to the device driver. When the temperature exceeds 95 °C, the ADC chip is disabled to avoid damaging the device.

## 1.3 External Inputs and Connectors

### 1.3.1 Front bracket inputs

The inputs of the Ndigo6G-12 board are located on the slot bracket.

Figure 1.2 shows the location of the four analog inputs A to D (see Section 1.3.3), the four digital TDC inputs 0 to 3 (see Section 1.3.4), and the two digital control inputs TRG and GATE (see Section 1.3.5).

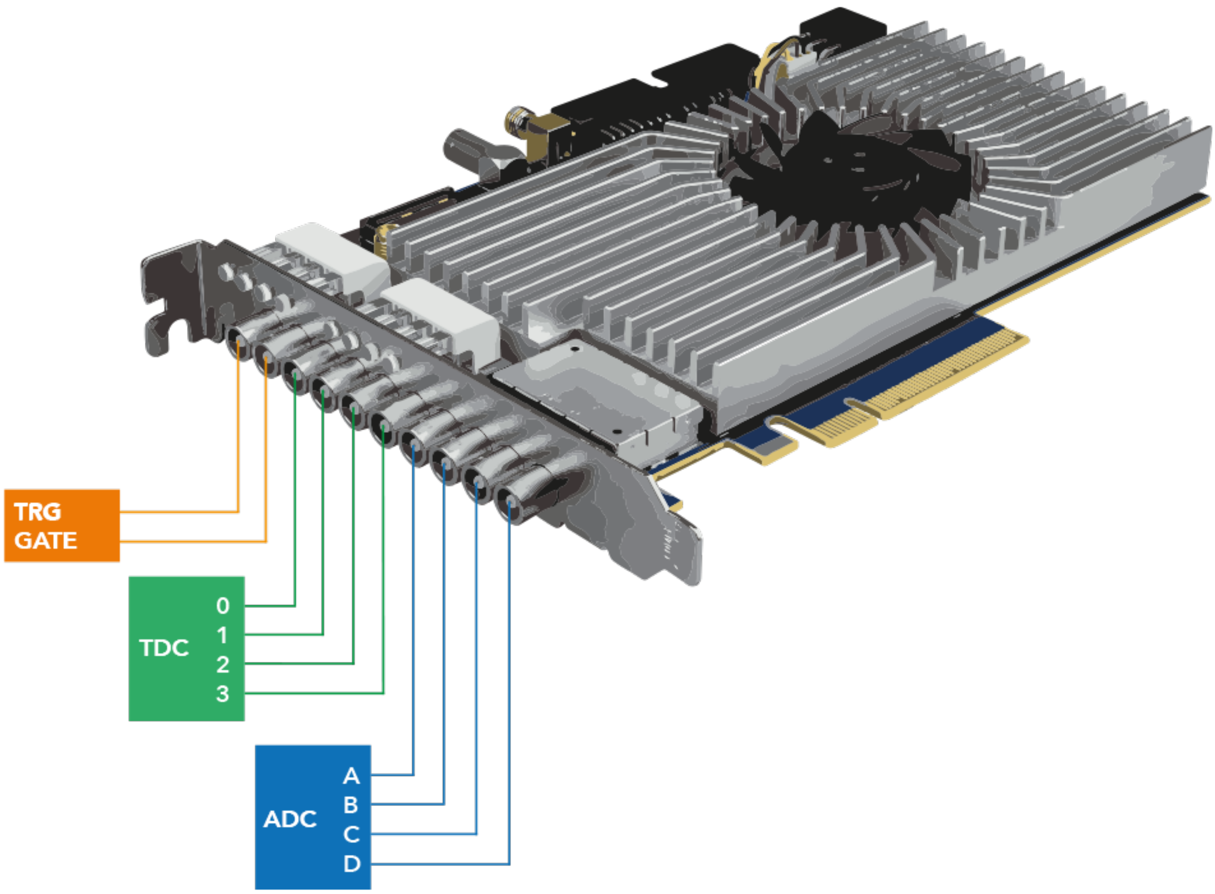



**Figure 1.2:** Input connectors of an Ndigo6G-12 board located on the PCI bracket.

### 1.3.2 Clock connections

Connectors to connect an external clock or to access the internal clock signal are located at the top of the board (see Figure 1.1).

**Clk in (SMA)**
: Connect your external 10 MHz clock signal here. Make sure to set *ndigo6g12_init_parameters::clock_source* to *NDIGO6G12_CLOCK_SOURCE_SMA*.

**Clk out (SMA)**
: 10 MHz output. This is either the internal clock signal, or an external clock 10 MHz clock if one is used.

**Clk diff (LEMO00)**
: Same as Clk out, but as a differential signal and with a LEMO00 connector.

### 1.3.3 Analog Inputs

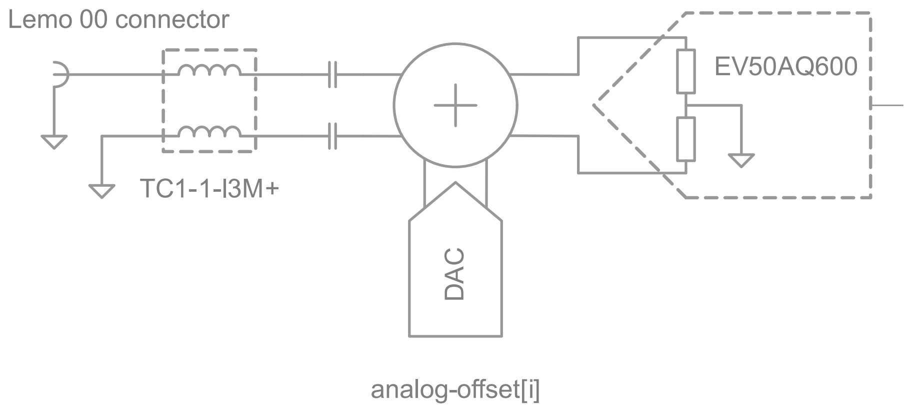



**Figure 1.3:** Input circuit for each of the four analog channels.

The analog inputs of the ADC are single ended LEMO00 coax connectors. The inputs have a 50 Ω impedance and are AC coupled. The inputs are converted to a differential signal using a balun.

#### *Analog Offsets*

AC coupling removes the DC voltage offset from the input signal. However, users can shift the DC baseline voltage before sampling to a value of their choice (using the `analog_offset` parameter).

This feature is useful for highly asymmetric signals, such as pulses from TOF spectrometers or LIDAR systems. Without analog offset compensation, the pulses would begin in the middle of the ADC range, effectively cutting the dynamic range in half (see Figure 1.5). By shifting the DC baseline to one end of the ADC range, the input range can be used fully, providing the maximum dynamic range. The analog offset can be set between ± 0.5 V.

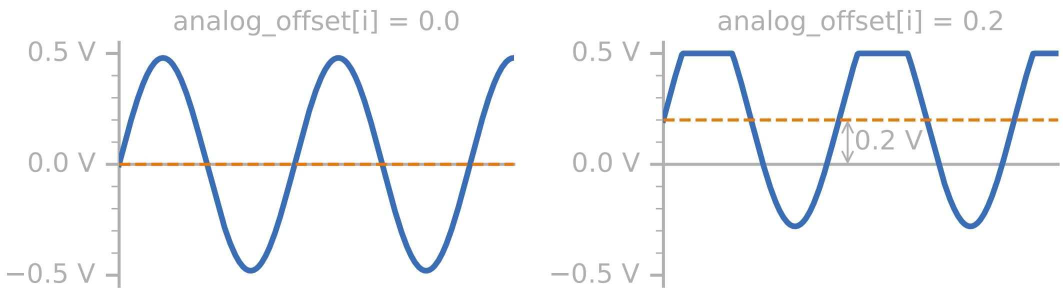



**Figure 1.4:** Users can add an analog offset to the input before sampling.

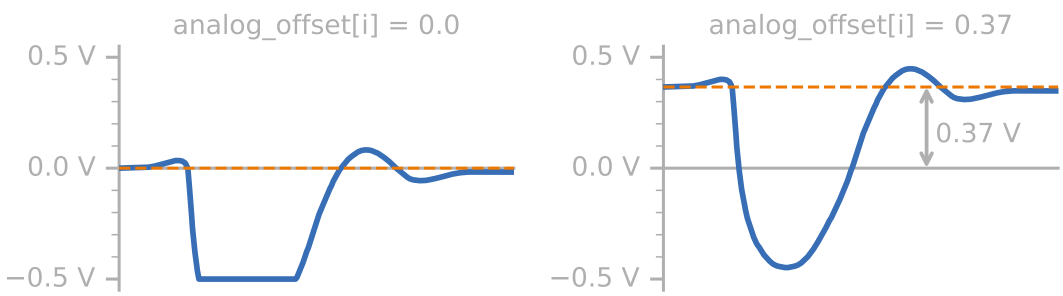



**Figure 1.5:** Asymmetric signal shifted to increase dynamic range.

### *AC-Coupling and Baseline Drift*

Due to the AC-coupling of the analog and digital inputs, a baseline drift may occur over time (see Figure 1.6). To avoid this problem, make sure your input signal fulfills the requirements laid out in Section 6.3.4.

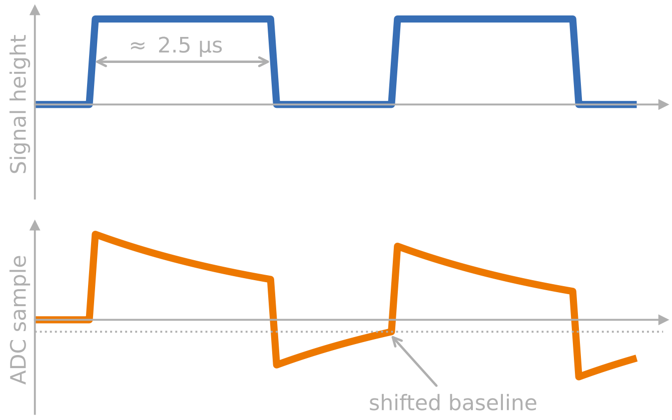



**Figure 1.6:** Baseline drift due to AC-coupling. A second pulse close to a first may be influenced by a shifted baseline, as sketched in the lower graph.

## 1.3.4 Digital TDC Inputs

The Ndigo6G-12 board includes four TDC channels with 13 ps timing resolution. The inputs are AC coupled (see Figure 1.7).

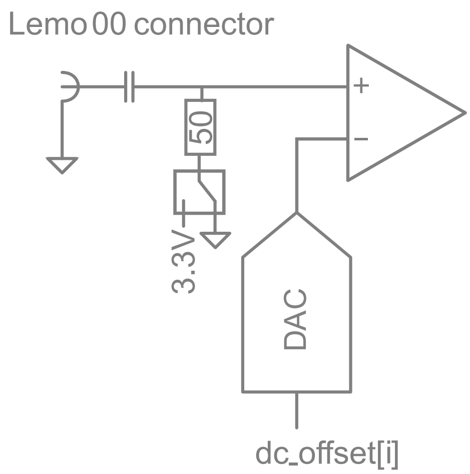



**Figure 1.7:** Principal input circuit for each of the digital TDC and control inputs.

The following members of the *ndigo6g12_configuration* struct configure, respectively, TDC channels 0 to 3:

*tdc_trigger_offsets[0:3]*
: Configure the DC offset.

*trigger[NDIGO6G12_TRIGGER_TDC0:NDIGO6G12_TRIGGER_TDC3]*
: Configure if an edge or level trigger is used (relevant, if the TDCs are used in *trigger_blocks* or *gating_blocks*) and if the rising or falling edge of the input signal triggers.

*tdc_configuration.channel[0:3]*
: Configure if (*channel[0:3].enable*) and when (*channel[0:3].gating_block*) timestamps

are recorded on the TDC channel.

The trigger unit input logic is summarized, as well, in Figure 2.14.

### 1.3.5 Digital Control Inputs

There are two digital control inputs on the front slot cover called TRG and GATE.

Input-signals on the inputs TRG and GATE are digitized and routed to the Trigger Matrix. They can be used to trigger any of the trigger state machines and *gating blocks* with maximum sampling rate.

The digital control inputs are optimally suited to be used as digital triggers and gates, and we recommend using them instead of the *digital TDC inputs* for these purposes.

TRG and GATE are configured analogously to the TDC inputs (see Section 1.3.4 and Figure 2.14), where indices 4 (5) and `NDIGO6G12_TRIGGER_TRG` (`NDIGO6G12_TRIGGER_GATE`) correspond to input TRG (GATE).

The input circuit and trigger logic is identical to the TDC inputs (see Figures 1.7 and 2.14).

#### *Use Control Inputs as TDCs*

The control inputs TRG and GATE can be used as low-resolution TDCs. The dead-time is 5 ns. Pulses should have a width of at least 300 ps to reliably be detected.

**Hint**

To record timestamps with the TRG or GATE input, set `config.tdc_configuration.channel[4||5].enable` to `true`.

**Note**

The digital *control* inputs TRG and GATE are best suited for triggering and controlling gates.

The digital *TDC* inputs are best suited for measuring precise time stamps.

# 2 Functionality

## 2.1 ADC Modes

The ADC quantizes the input signal using 12 bits. By default, these are mapped to signed 16 bit (for more details, see Section 4.3).

Data processing such as trigger detection or packet building are always performed at 5 ns intervals. Depending on the ADC mode, this interval may contain 32 (*1-Channel Mode* @ 6.4 Gsps), 16 (*2-Channel Mode* @ 3.2 Gsps) or 8 (*4-Channel Mode* @ 1.6 Gsps) samples.

The ADC mode is configured using *ndigo6g12_configuration::adc_mode*.

The board supports using one, two or four channels. This is configured when the board is initialized, see *ndigo6g12_init_parameters::application_type*.

During interleaving, the Ndigo6G-12 firmware reorders and groups the data into a linear sample stream. The process is fully transparent. For users, the only difference is that a 5 ns cycle can contain 8, 16 or 32 samples, depending on the mode.

Depending on the *application_type*, the minimal length of the output packets changes. The minimal lengths are:

- 3 × 32 Samples (15 ns) @ 6.4 Gsps (1-Channel Mode)
- 3 × 16 Samples (15 ns) @ 3.2 Gsps (2-Channel Mode)
- 4 × 8 Samples (20 ns) @ 1.6 Gsps (4-Channel Mode)

### 2.1.1 1-Channel Modes A and D

In these modes, only a single channel is used. The analog signal on that channel is digitized at 6.4 Gsps. Packet size is always a multiple of 32 samples per 5 ns (See Figures 2.1 and 2.12).

For this mode, *ndigo6g12_static_info::application_type* needs to be either *NDIGO6G12_APP_TYPE_1CH* or *NDIGO6G12_APP_TYPE_AVRG*.

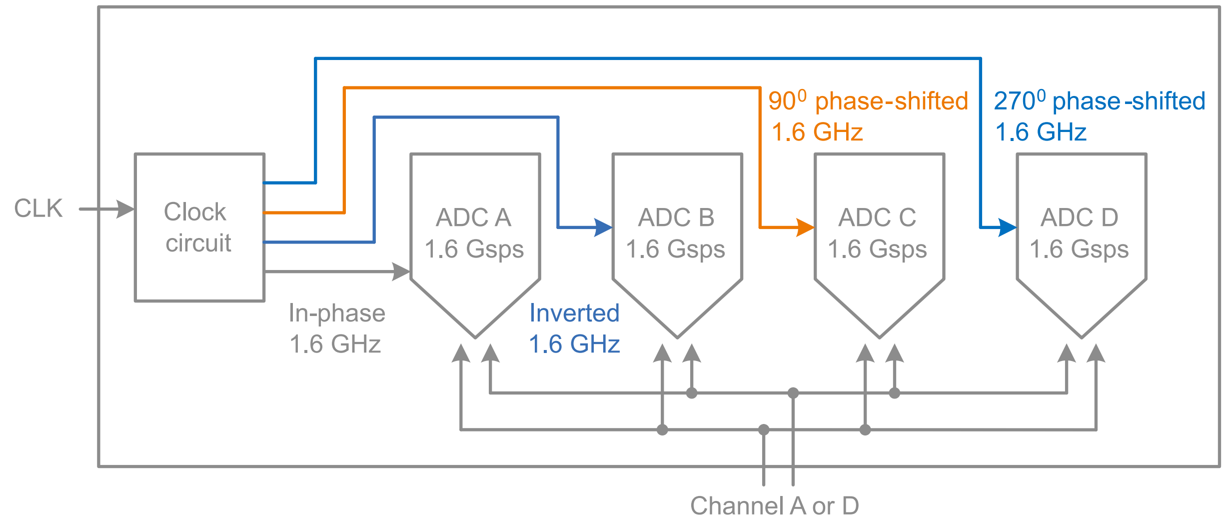



**Figure 2.1:** ADCs in 1-channel-mode A, B, C or D interleaved for 6.4 Gsps.

### 2.1.2 2-Channel Mode AD

In this mode, two channels are used simultaneously. The analog signals on these channels are digitized at 3.2 Gsps each. Packet size is always a multiple of 16 samples per 5 ns (See Figures 2.2 and 2.11).

For this mode, *ndigo6g12_static_info::application_type* needs to be *NDIGO6G12_APP_TYPE_2CH*.

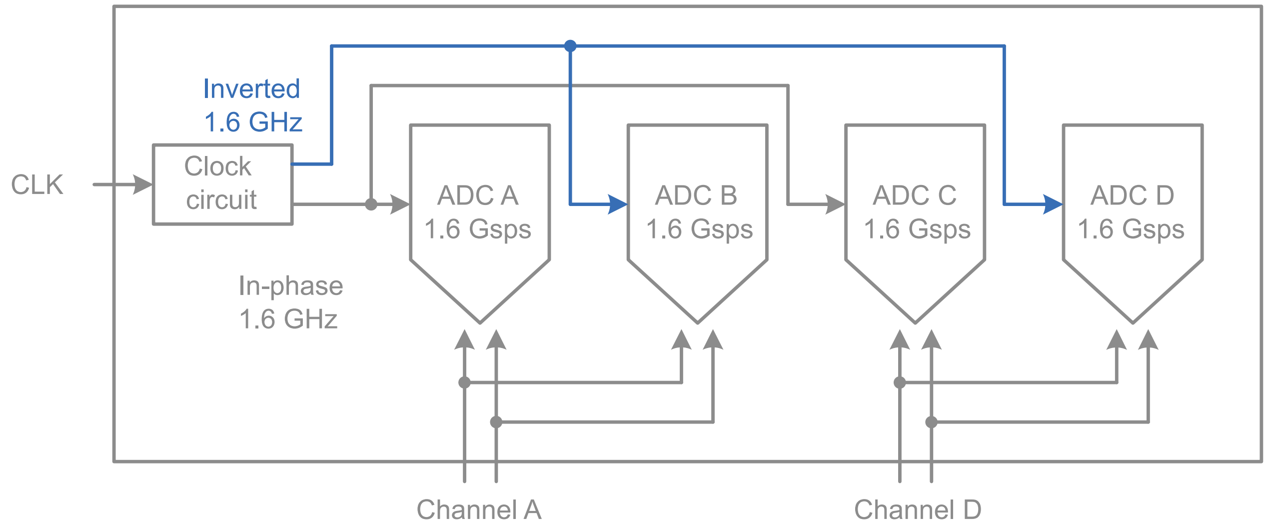



**Figure 2.2:** ADCs in 2-channel-mode AD, interleaved for 3.2 Gsps.

### 2.1.3 4-Channel Mode ABCD

In this mode, all four channels are digitized independently at 1.6 Gsps each. The packet size is always a multiple of 16 samples per 10 ns. (See Figures 2.3 and 2.10).

For this mode, *ndigo6g12_static_info::application_type* needs to be *NDIGO6G12_APP_TYPE_4CH*.

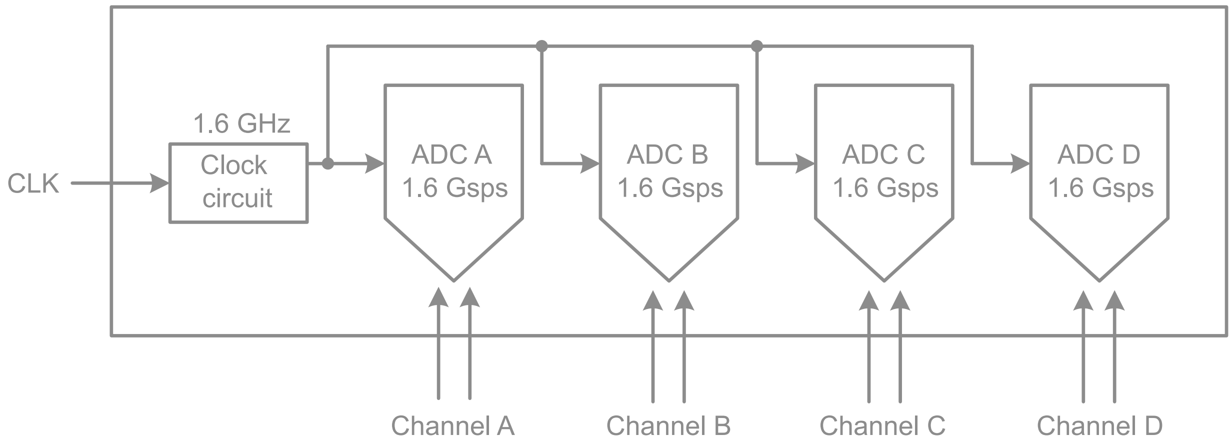



**Figure 2.3:** ADCs in 4-channel-mode ABCD at 1.6 Gsps.

### 2.1.4 Multiple Sampling Modes

In these modes, only the specified input channels are used, but the channels are sampled independently by the ADC cores. The output of the board depends on *ndigo6g12_configuration::sample_averaging*.

- `sample_averaging == false`: The digitized samples are output as separate packets (the number of which depends on the selected mode).

- `sample_averaging == true`: The average of the digitized samples is calculated and output as one single packet.

Using the same trigger settings on all ADCs can be used to reduce noise by averaging the four channels. To deal with complex triggering conditions, different trigger settings on each of the ADCs can be used.

The Ndigo6G-12 provides four ADCs sampling at 1.6 Gsps each. Higher speed modes are implemented by interleaving two or four of these ADCs.

### Modes AA and DD

In this mode, input channel A (or D) is sampled at 3.2 Gsps two times and independently by the internal ADC cores, see Figure 2.4.

For this mode, *ndigo6g12_static_info::application_type* needs to be *NDIGO6G12_APP_TYPE_2CH*.

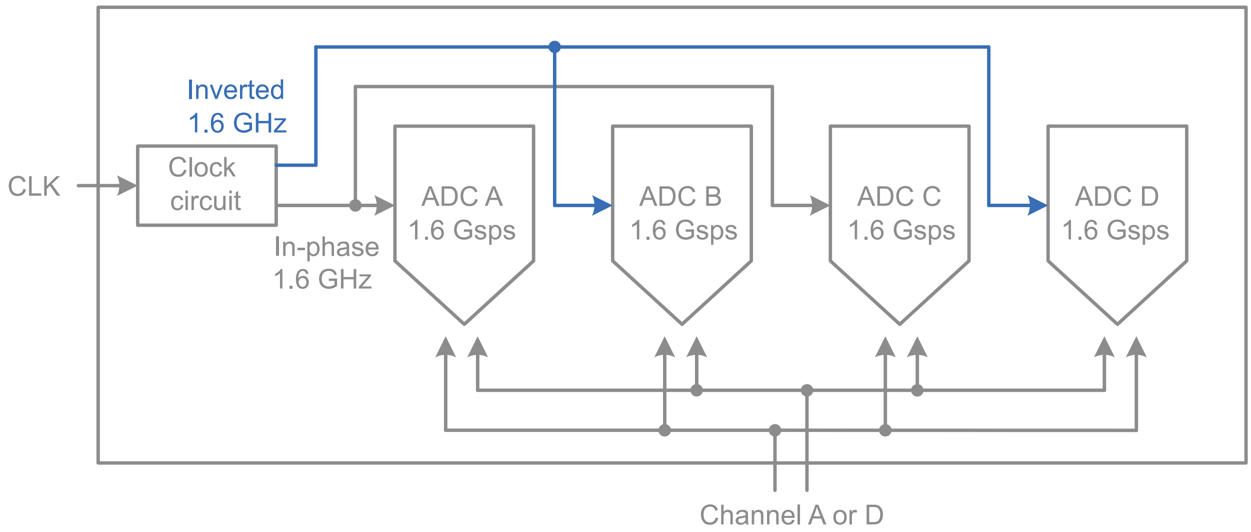



**Figure 2.4:** ADCs in 2-channel-mode AA or DD at 3.2 Gsps.

### Mode AADD

In this mode, input channel A and D are sampled at 1.6 Gsps two times and independently by the internal ADC cores, see Figure 2.5.

For this mode, *ndigo6g12_static_info::application_type* needs to be *NDIGO6G12_APP_TYPE_4CH*.

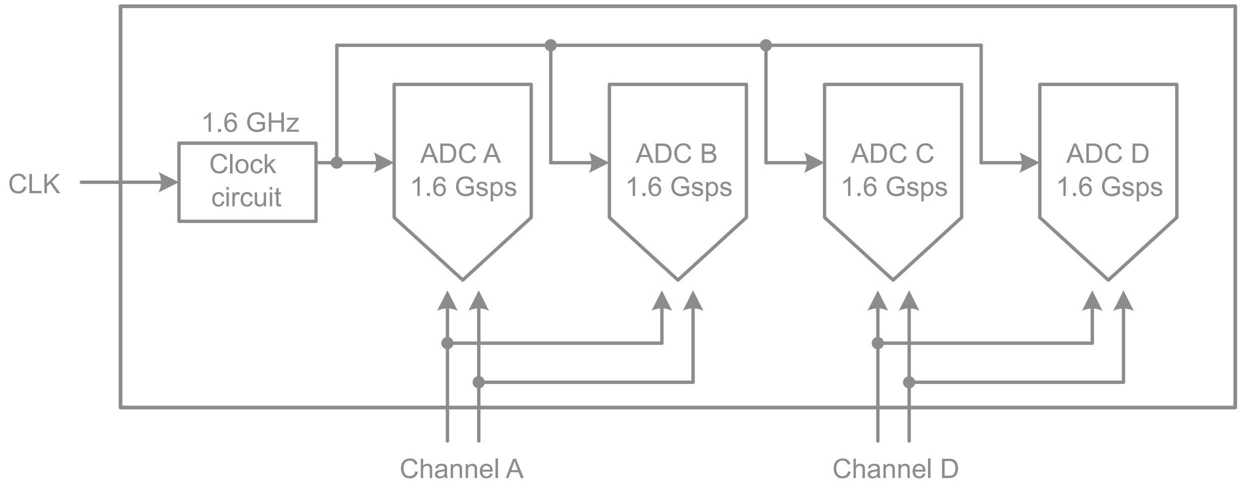



**Figure 2.5:** ADCs in 4-channel-mode AADD at 1.6 Gsps.

#### *Modes AAAA, DDDD*

In this mode, input channel A (or D) are sampled at 1.6 Gsps four times and independently by the internal ADC cores, see Figure 2.6.

For this mode, *ndigo6g12_static_info::application_type* needs to be *NDIGO6G12_APP_TYPE_4CH*.

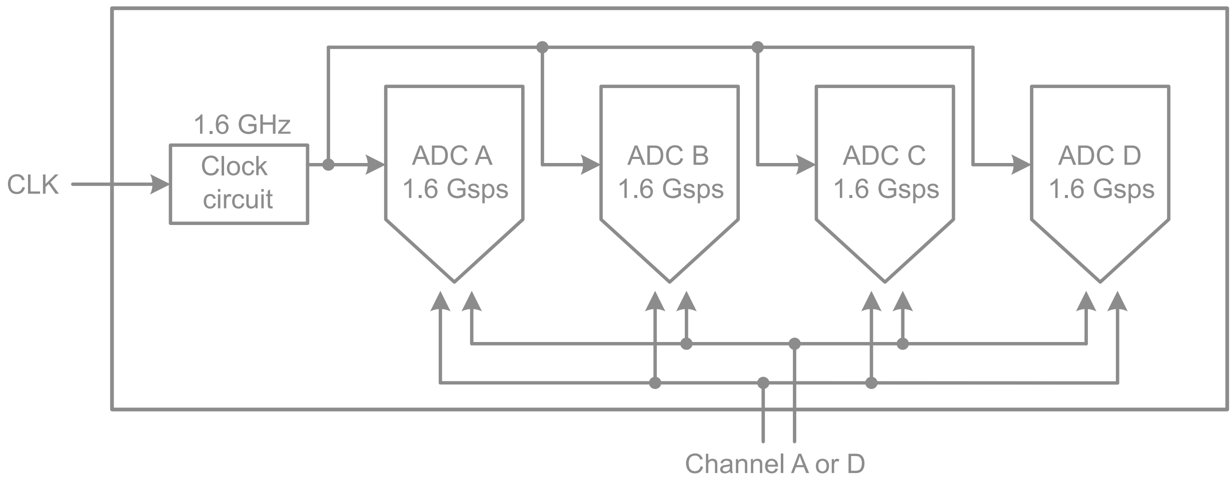



**Figure 2.6:** ADCs in 4-channel-mode AAAA or DDDD at 1.6 Gsps.

## 2.2 Zero Suppression

One of the Ndigo6G-12's key features is on-board zero suppression to reduce PCIe bus load. Only data that passes specifications predefined by the user is transmitted. Data is transmitted as so-called "*packets*." For the ADC channels, the packet contains the waveform data and a timestamp giving the absolute time (i.e., the time since the start of the data acquisition) of the packet's first sample.

Figure 2.7 shows a simple example: Data is only written to the PC if the sample values exceed a specific threshold. Expanding on that, the Ndigo6G-12's zero suppression can be used to realize much more complex scenarios using the *Trigger* and *Gating Blocks* (see Sections 2.3 and 2.4).

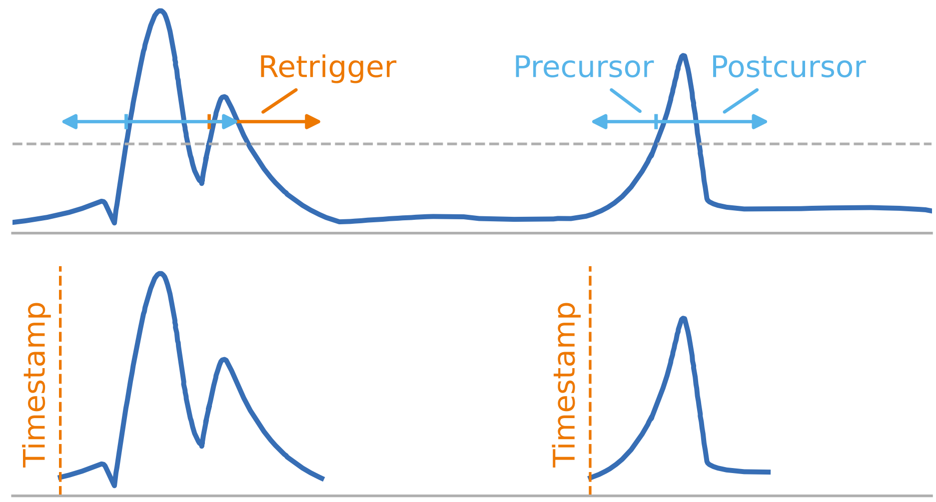



**Figure 2.7:** Simple zero suppression: Only data with values above a threshold are written to the PC.

## 2.3 Trigger Setup

The Ndigo6G-12 records analog waveforms using zero suppression. Whenever a relevant waveform is detected, data is written to an internal FIFO memory.

Each ADC channel has two *trigger units*. These can be configured independently (e.g., one unit could trigger on rising edges, the other on falling). They are configured with *config.trigger*.

Each ADC channel has a corresponding *trigger block* that determines whether data is written to the internal FIFOs. The trigger blocks are configured with *config.trigger_block*. Each trigger block can take any amount of trigger units as a source (for details, see *ndigo6g12_trigger_block::sources* or Section 2.3.2), thus, enabling sophisticated trigger setups.

### 2.3.1 Trigger configuration

#### *Analog Inputs*

Users can specify a *threshold* and can choose whether triggering is used whenever incoming data is below or above the threshold (level triggering, see Figure 2.8) or only if data exceeds the threshold (edge triggering, see Figure 2.9).

A gate *length* can be set to extend the recording window by multiples of 5 ns. Furthermore, a *precursor* window can be specified, causing the trigger unit to write data to the FIFO (precursor × 5 ns) before the trigger event.

When edge triggering is used, all packets have the same length of (precursor + length + 1)-cycles of 5 ns. For level triggering, packet length is data dependent.

If *retrigger* is enabled and the trigger conditions are fulfilled during the recording of the postcursor, the recording window is extended (see Figure 2.7).

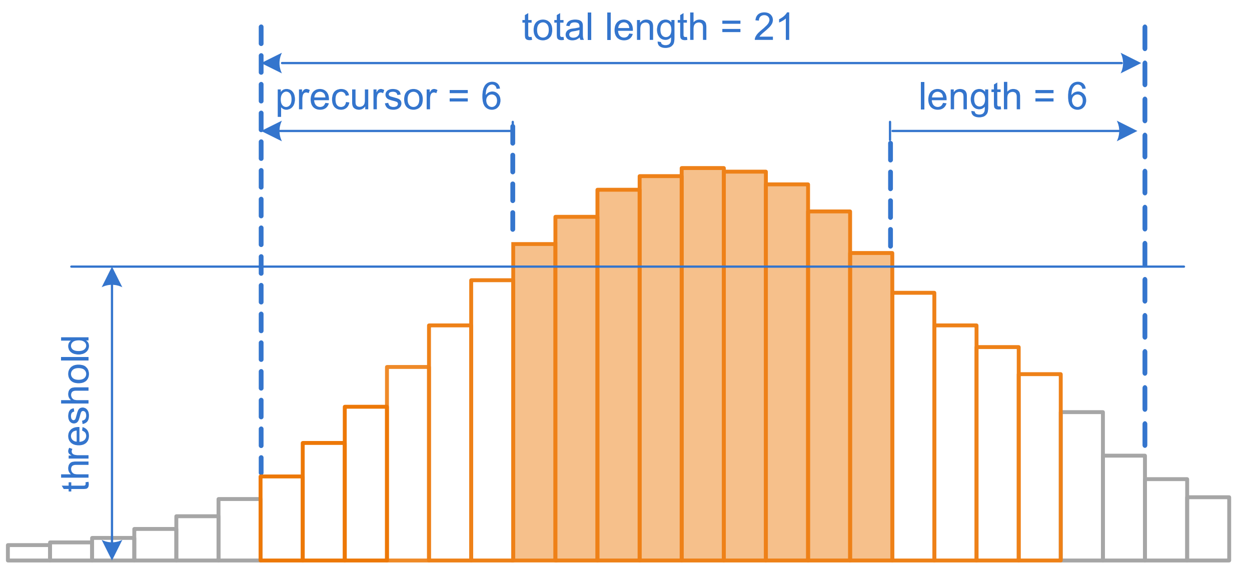



**Figure 2.8:** Example for level triggering.

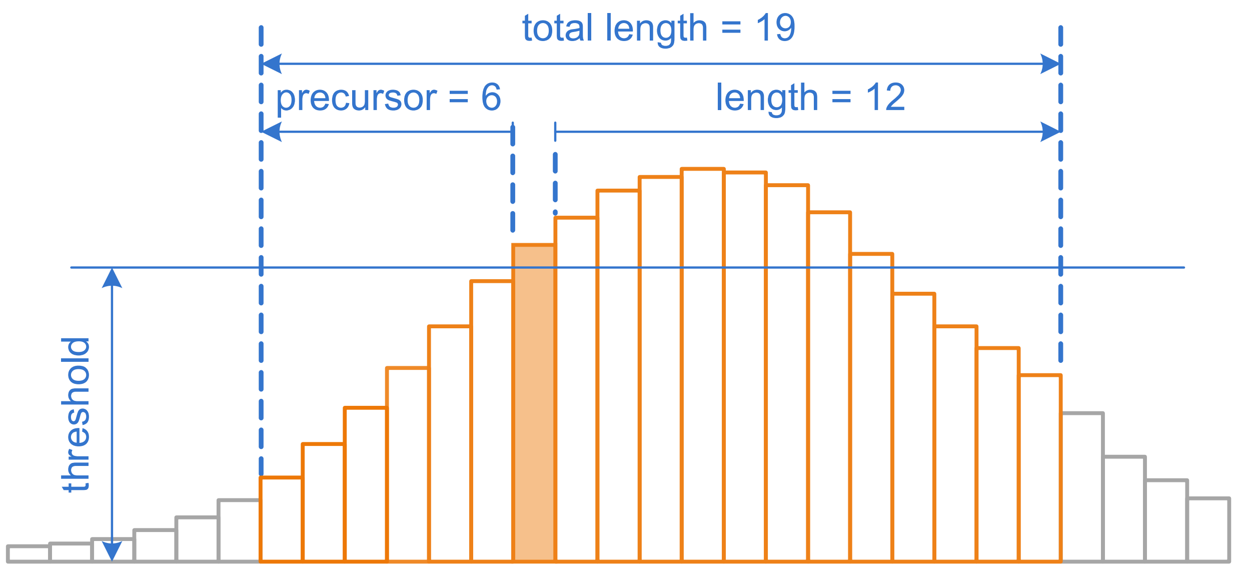



**Figure 2.9:** Example for edge triggering.

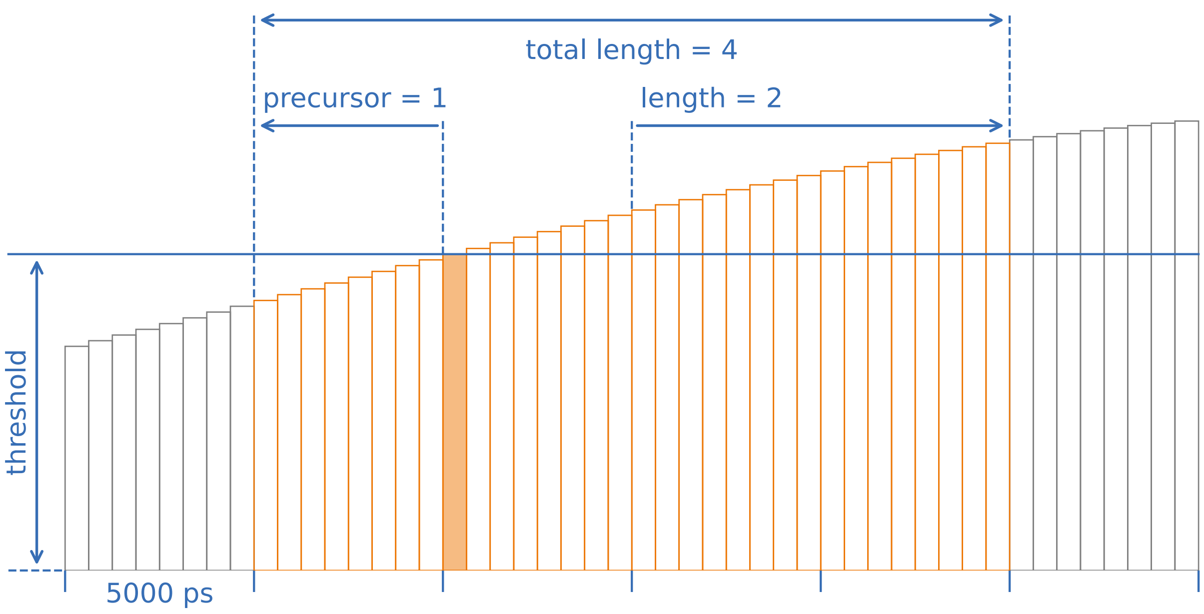



**Figure 2.10:** Triggering in 4-channel mode at 8 samples per clock cycle.

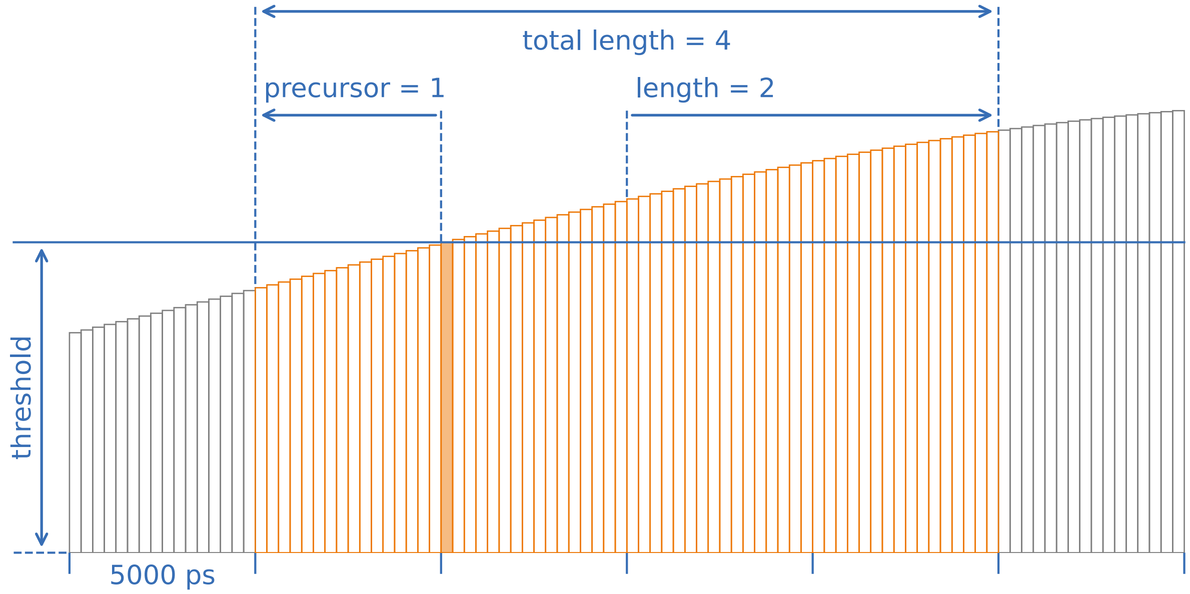



**Figure 2.11:** Triggering in 2-channel mode at 16 samples per clock cycle.

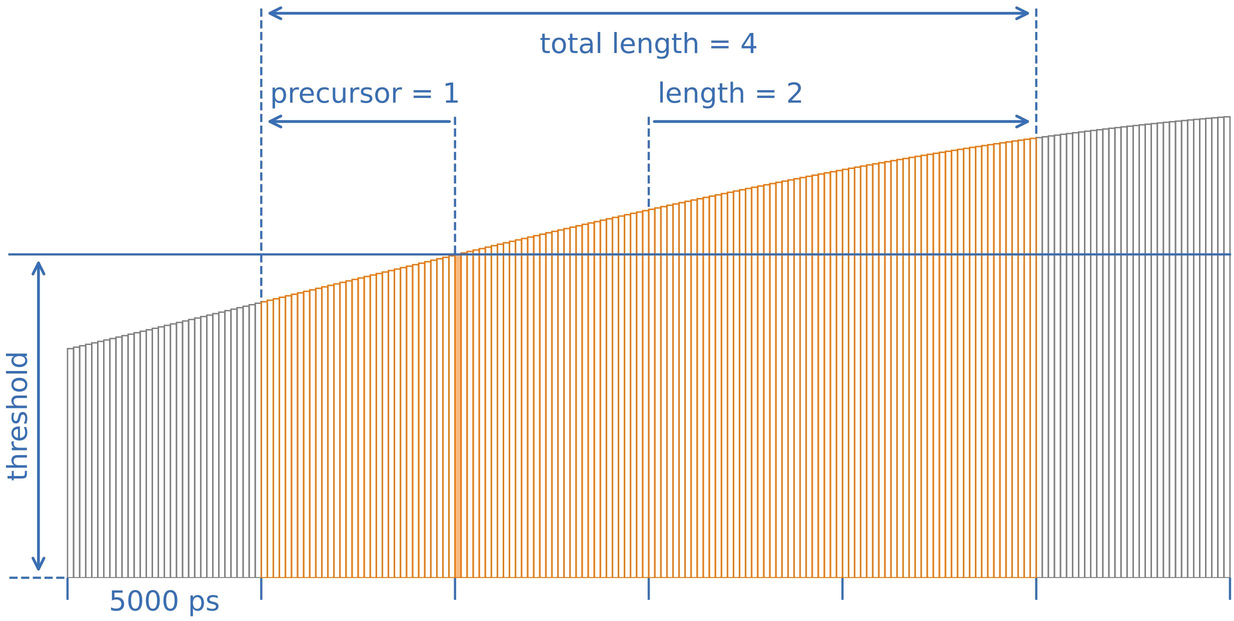



**Figure 2.12:** Triggering in 1-channel mode at 32 samples per clock cycle.

#### *Digital Inputs*

For all digital inputs, the configuration value *ndigo6g12_trigger::threshold* is ignored. Their trigger threshold is configured by *ndigo6g12_configuration::tdc_trigger_offsets*.

Equivalently to the analog inputs, edge- or level-trigger functionality can be enabled using *ndigo6g12_trigger::edge*. The duration of a level trigger is solely limited by the AC-coupling (see Figure 1.6 for the effects of AC-coupling on a signal).

### 2.3.2 Trigger inputs

A *trigger_block* can use several input *sources*:

- The eight trigger decision units of all four ADC channels (Figure 2.13)
- The four TDC and the two digital control inputs (Figure 2.14)
- A function trigger providing random or periodic triggering (see *Auto Triggering Function Generator*).

Trigger inputs from the above sources can be concatenated using a logical OR by setting the appropriate bits in the bitmask (see *ndigo6g12_trigger_block::sources*).

See also Figure 2.15.

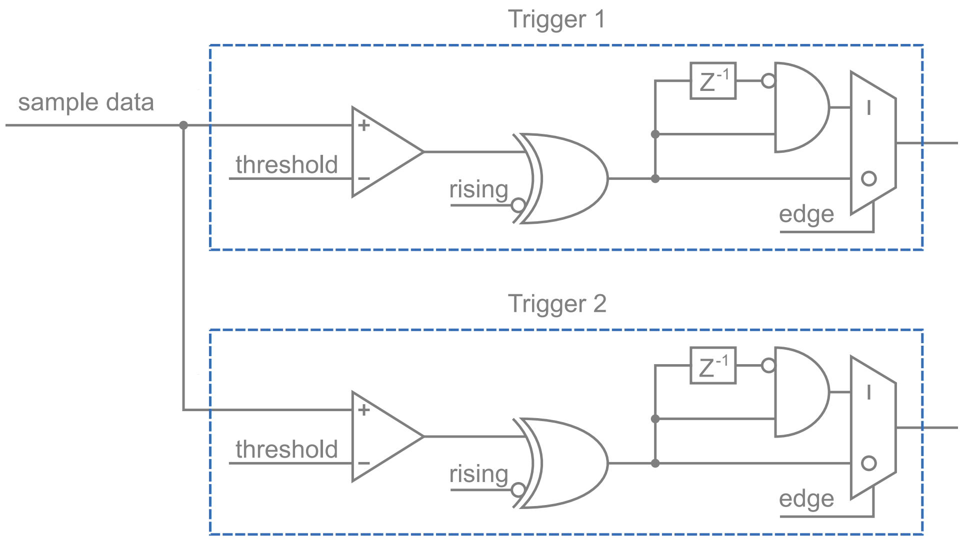



**Figure 2.13:** From the ADC inputs, a trigger unit creates an input flag for the trigger matrix. Each digitizer channel (A, B, C, D) has two trigger units.

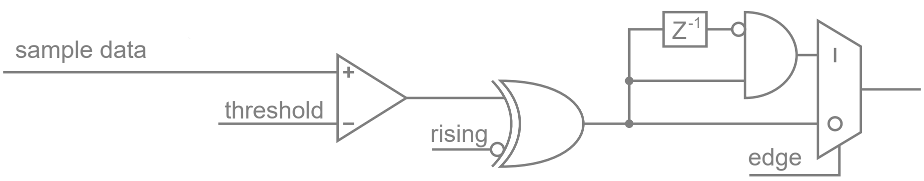



**Figure 2.14:** The digital inputs TDC0, TDC1, TDC2, TDC3, TRG, and GATE have simpler trigger units.

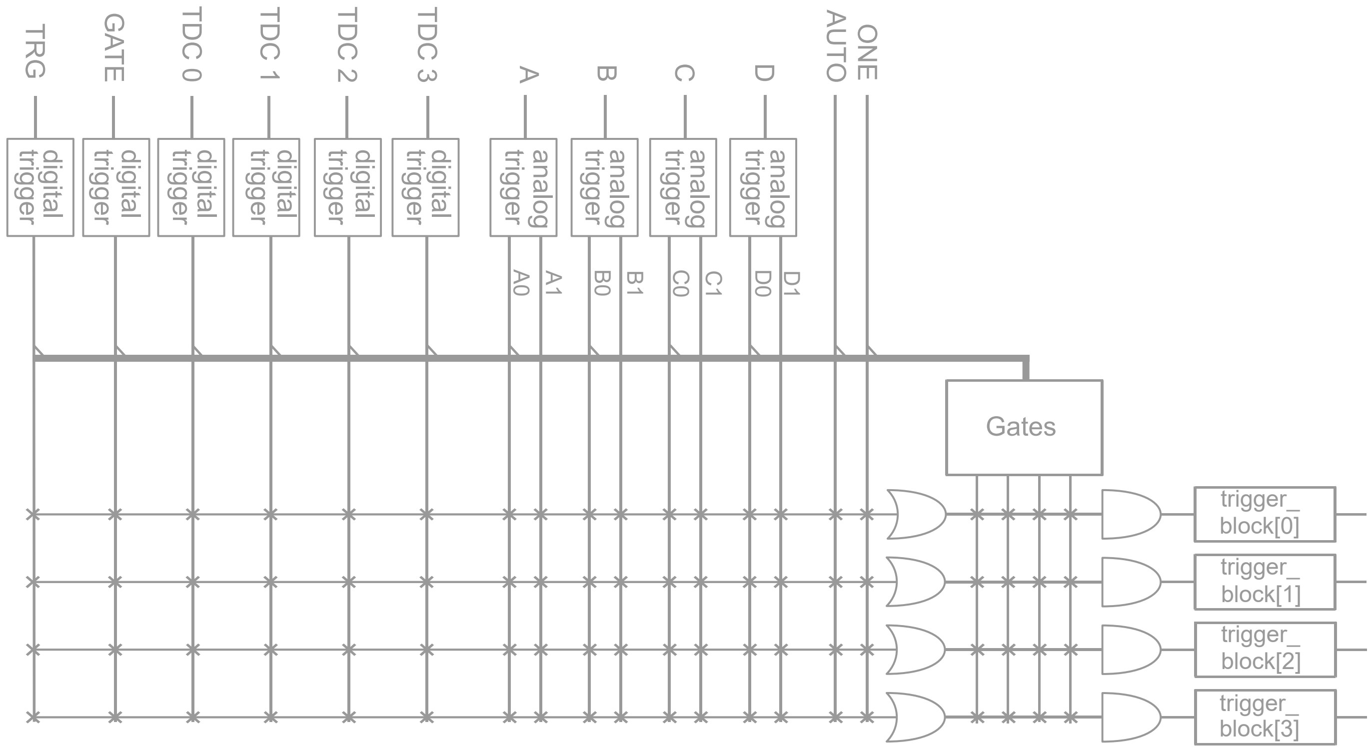



**Figure 2.15:** Trigger Matrix. The eight trigger signals from the four analog channels and the trigger signals from the six digital channels (four TDC channels, TRG, GATE) can be combined to create a trigger input for each *trigger block*. Additionally, four *gate signals* (see Figure 2.16) can be used to suppress trigger during configurable time frames.

### 2.3.3 Gating trigger events

Triggers can be fed into the *gating_blocks* as outlined in Chapter 2.4 and Figure 2.16.

In return, the *gating_blocks* can be used to block writing data to the FIFO. That way, only zero-suppressed data occurring when the selected gate is active is transmitted. This procedure reduces PCIe bus load even further.

Which *gating_block* is used to block a particular *trigger_block* is configured with *ndigo6g12_trigger_block::gates*.

## 2.4 Gating Blocks

In order to decrease the amount of data transmitted to the PC, the Ndigo6G-12 includes four independent gate and delay units.

They are configured using *ndigo6g12_configuration::gating_block* and (specifically for the TDC channels) *ndigo6g12_tdc_channel::gating_block*.

A gate and delay unit creates a gate window starting and closing at specified times after a trigger event (as configured by the user with *ndigo6g12_gating_block::start* and *stop*).

Concretely, if a trigger event is detected, a timer starts. After the timer reaches the time corresponding to *start*, the gate will activate. After the timer reaches the time corresponding to *stop*, it will inactivate.

This behavior may be influenced by the *retrigger* feature. With this feature enabled, another trigger signal will reset the timer to zero. That means, if a second trigger is detected *before* the gate is activated,

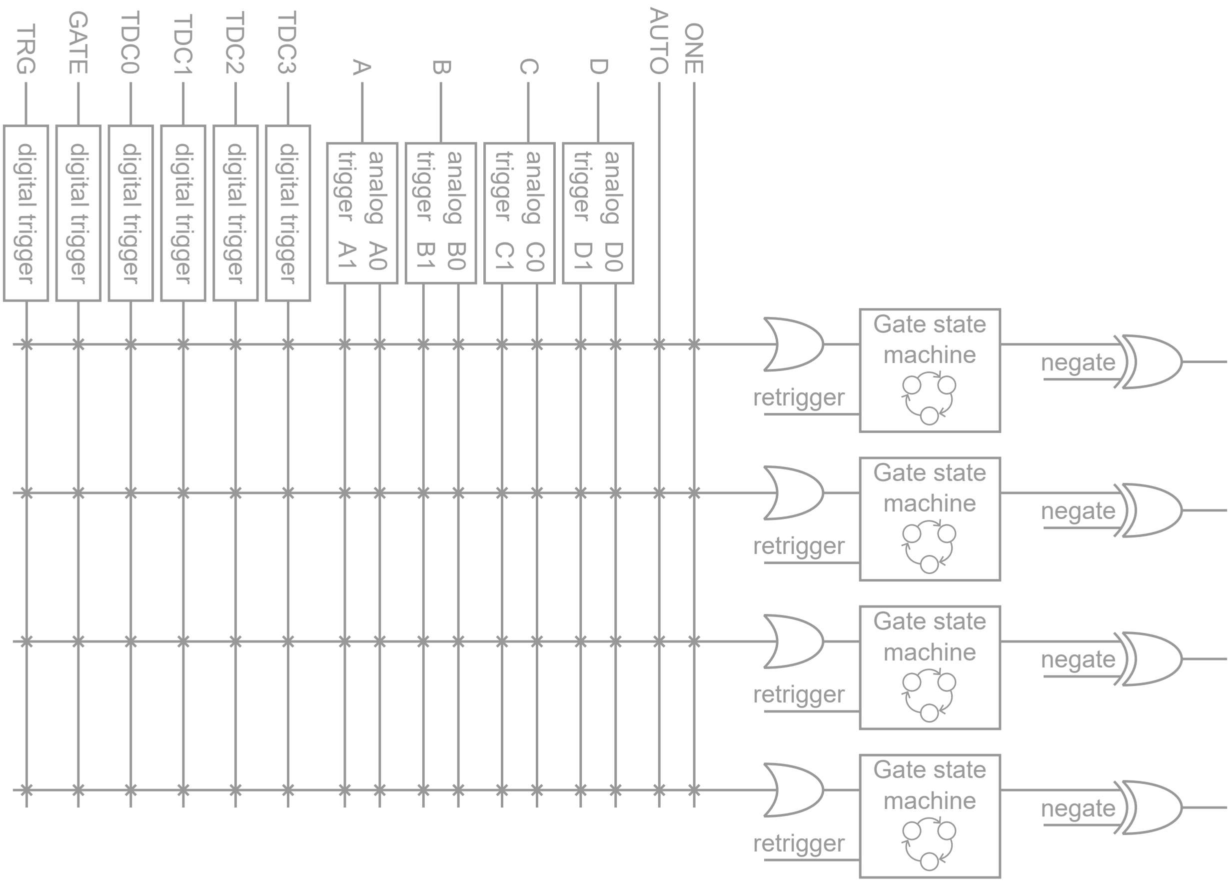



**Figure 2.16:** Gating Blocks: Each gating block can use an arbitrary combination of inputs to trigger its state machine. The outputs can be individually inverted and routed to the AND-gate feeding the trigger blocks.

the time until it *activates* is extended. If, however, the gate was already active, the time until it *inactivates* will be extended.

> **Attention**
>
> A bug in Firmware Rev. ≤*1.24120* causes the `retrigger` feature to reset the gate logic entirely (i.e, the state of the gate will inactivate after a retrigger event).

Depending on *ndigo6g12_gating_block::negate*, an active gate will be open (signal detection enabled) or closed (signal detection disabled).

Each gating block can use an arbitrary combination of inputs which trigger it. This is configured using *ndigo6g12_gating_block::sources*.

*trigger_blocks* can use the gate signal to suppress data acquisition, that is, only data that fulfills zero suppression specifications occurring in an open gate window is written to the PC.

Figure 2.17 shows the functionality of the gate timing and delay unit.

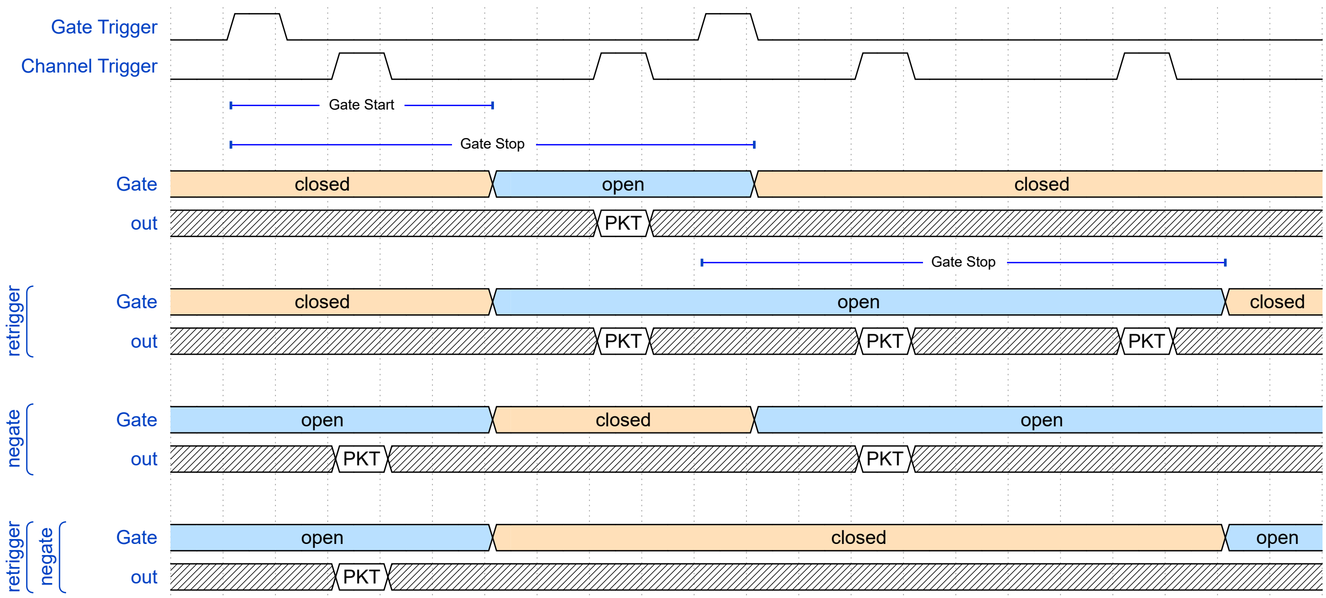



**Figure 2.17:** Gate and delay functionality: When a trigger occurs, the gate opens after a set period of time "Gate Start" and closes when it reaches "Gate Stop". A second trigger event may influence this behavior if retriggering is enabled.

### 2.4.1 Examples

***Example 1: Suppression of Noise After Starting an Acquisition***

In mass spectrometer and other experiments, noise while starting data acquisition can result in undesired trigger events during start-up time. To prevent noise in the output data, a gating block could be used to suppress all triggers during start-up.

The following example illustrates the use of a gating block (in the following, *gating_block[0]*) to prevent recording noise:

- Set up the GATE input to trigger on each acquisition start, that is, *trigger[NDIGO6G12_TRIGGER_GATE]* is configured corresponding to the input signal (e.g., configuring the polarity).

- *NDIGO6G12_TRIGGER_SOURCE_GATE* is selected as input source of *gating_block[0].source* and the *gating_block[0].start* parameter is set to 0.
- The *gating_block[0].stop* parameter is set to the desired length (in multiples of 5 ns).
- *gating_block[0].negate* is set to `true`.

Now, *gating_block[0]* will output a LOW pulse of the desired length (that is, the gate is closed during start-up time) whenever there is a pulse on the GATE input.

Now, select the above gate for the trigger bock you want to use for triggering data acquisition, e.g., *trigger_block[0]*:

- Set *trigger_block[0].sources* e.g.,

  ```
  config.trigger_block[0].sources = NDIGO6G12_TRIGGER_SOURCE_A0 | NDIGO6G12_
    ↪TRIGGER_SOURCE_D0
  ```

  uses the ADC input channels A and D as sources.
- Set *NDIGO6G12_TRIGGER_GATE_0* as *trigger_block[0].gates*.

  ```
  config.trigger_block[0].gates = NDIGO6G12_TRIGGER_GATE_0
  ```

Now, recording of data is suppressed for an initial start-up time.

### 2.4.2 Example 2: Delayed Trigger

To sample a short window at a specified time after a trigger event on a channel, a gating block can be used to create a delayed trigger. To do this, one of the triggers of the channel of interest is configured to the desired parameters by selecting the threshold, setting the edge polarity and enabling edge triggering.

Instead of directly using this trigger as an input to the trigger block's input matrix, the trigger is selected as an input to a gating block. The block is configured with `start = delay` (in multiples 5 ns) and `stop = start+1`, `negate = false`. This causes the gating block to produce a one clock cycle pulse on its output after the specified delay.

To send this pulse to the trigger block, the gating block must be enabled in the trigger block's AND matrix and the ONE trigger source must be selected.

## 2.5 Auto Triggering Function Generator

Some applications require periodic or random triggering. The Ndigo6G-12's 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 given by the exponent $N$.

The period is

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

clock cycles with a duration of 5 ns per cycle, where $6 \leq M < 2^{32}$ and $0 \leq M < 32$.

This allows to monitor input signals at times the current trigger configuration does not trigger, e.g., to get baseline information in mass spectrometry applications. It can also be used to determine a suitable threshold level for the trigger by first getting random statistics on the input signal.

This functionality is enabled and configured using *ndigo6g12_configuration::auto_trigger_period* and *auto_trigger_random_exponent*.

## 2.6 Averaging Mode

Instead of streaming each recorded trigger event as packets, it is possible to average over multiple trigger events.

By initializing the Ndigo6G-12 board with *NDIGO6G12_APP_TYPE_AVRG*, Averaging Mode is enabled. Then, a number of *ndigo6g12_averager_configuration::iterations* are averaged before output is written.

Averaging Mode can be used only with ADC modes A and D (see Section 2.1).

> **Attention**
>
> Be aware that in averaging mode, the first two 64-bit words in *data* are an extended header. See Section 4.5 for more information.

## 2.7 Timing Generator (TiGer)

The LEMO connectors of all TDC channels, the TRG channel, and the GATE channel can be used as an AC-coupled trigger output. The TiGer functionality can be configured independently for each connector.

Each TiGer is configured using the *ndigo6g12_tdc_tiger_block* struct. The tiger blocks can be triggered by any combination of inputs, including the auto-trigger and the ADC channels.

> **Note**
>
> The TiGer configuration is similar to the *gating blocks*.

The TiGer can be used in different output *modes*. For an overview of the different modes, see the documentation in the API section.

With restrictions, the respective LEMO connectors can be used simultaneously as a TiGer output *and* as an input.

## 2.8 Performing a firmware update

The Ndigo6G-12 device driver includes the tool *FirmwareGUI_64.exe*. It can be used to perform a firmware update.

The tool is located in your device driver installation path under `apps\x64\`.

The tool is shown in Figure 2.18.

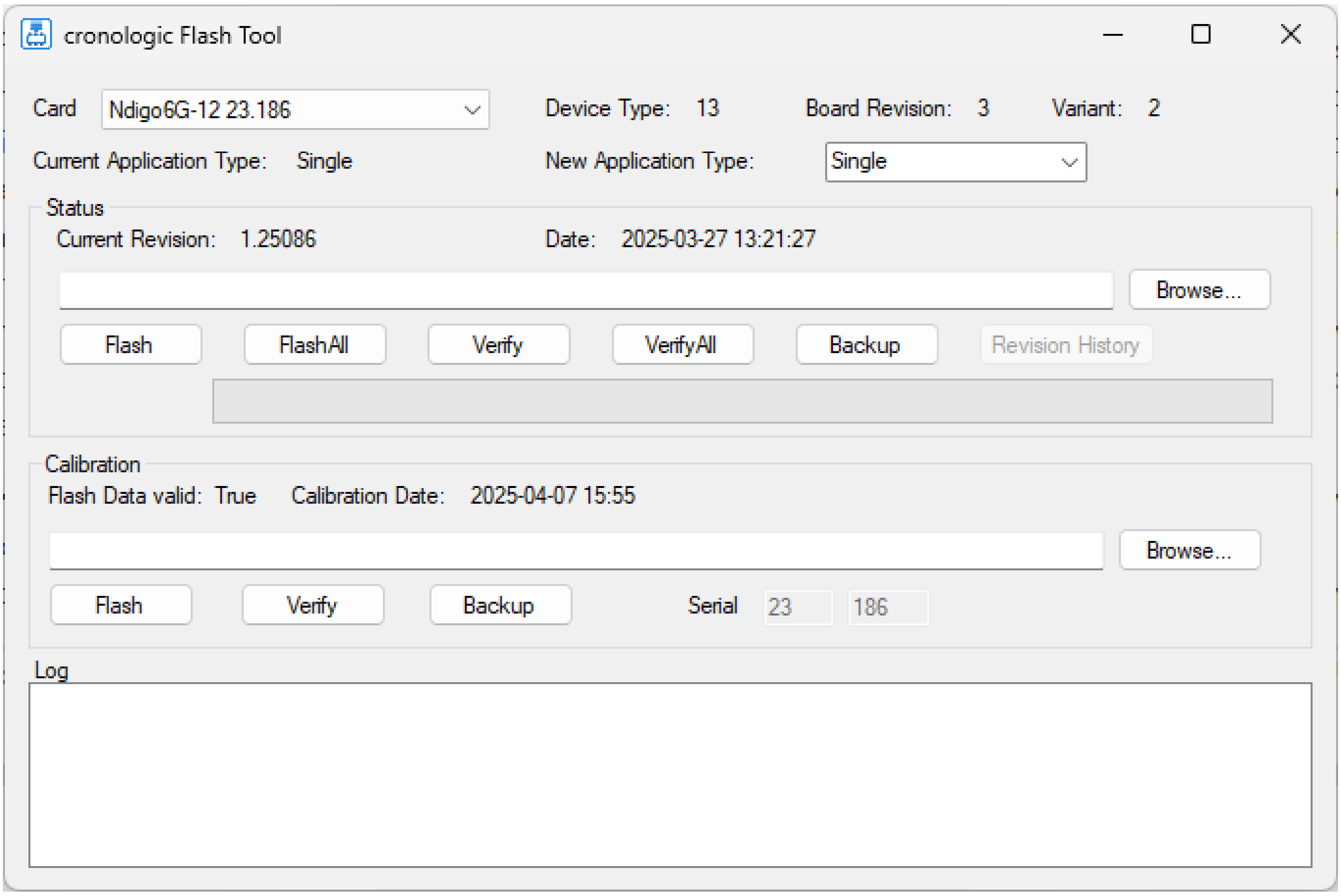



**Figure 2.18:** Firmware flash tool for the Ndigo6G-12.

### 2.8.1 Procedure

1. If you have multiple Ndigo6G-12 installed, choose a card in the "Card" dropdown menu.
2. It is advisable to perform a backup of the current firmware and calibration data. Click the respective "Backup" button in the tool.
3. Optionally select the application type that you wish to use. This step is not required and can also be performed when configuring the Ndigo6G-12 in your user software.
4. Browse to the new firmware file: In the "Status" section, click on "Browse". The firmware that is delivered with the device driver is located at `firmware\Ndigo6G_Firmware_YYYYMMDD.cronorom`.
5. Optionally compare the currently installed firmware with the selected firmware by pressing "Verify".
6. Click "Flash" to perform the firmware update.

**Attention**

The new firmware will only be used after a *complete* power cycle. A simple reboot may not be sufficient.

**Attention**

After a firmware update the TDCs have to be re-calibrated. See *Calibrating the TDC* for the procedure.

## 2.9 Calibrating the TDC

You can backup or restore the calibration from a previous backup using *FirmwareGUI_64.exe* (see *Performing a firmware update* and Figure 2.18).

In the "Calibration" section, click "Backup" and choose a location for the `.ndigo6gcal` calibration file.

You can compare the currently used calibration with the calibration from a backup file by clicking "Verify" button after browsing to your calibration file.

You can flash the calibration from a `*.ndigo6gcal` file onto the Ndigo6G-12 by clicking "Flash" after browsing to your calibration file.

> **Attention**
>
> After performing a *firmware update*, it is always necessary to re-calibrate your Ndigo6G-12. Restoring a previous calibration from a backup is not sufficient.

### 2.9.1 Re-calibrating the Ndigo6G-12

Calibration is performed with the command-line tool *ndigo6g12_tdc_alignment.exe*. It is located in the installation directory of the Ndigo6G-12 driver under `apps\x64` (by default `C:\Program Files\cronologic\Ndigo6G-12\apps\x64`).

Navigate to the folder and open it in a terminal, then start the tool from that terminal.

The tool takes command line arguments. Run `.\ndigo6g12_tdc_alignment.exe -help` for an overview.

Calibration is performed by simply starting the tool: `.\ndigo6g12_tdc_alignment.exe`.

If multiple Ndigo6G-12 are installed in your system, you can perform calibration for each one of them by calling `.\ndigo6g12_tdc_alignment.exe -index <device_index>`, where the `<device_index>` starts at 0 and increments for each Ndigo6G-12.

# 3 Driver Programming API

**Attention**

The API requires *driver versions* **>2.0.0** and *firmware* **1.24120**.

The API is a DLL with C linkage. Declarations of the interface are found in `ndigo6g12_interface.h`, provided by the Ndigo6G-12 driver.

This chapter provides an overview of the provided API functionality.

## 3.1 Constants

### 3.1.1 General

`NDIGO6G12_API_VERSION`

The current API version.

`NDIGO6G12_TRIGGER_COUNT`

The number of ADC and TDC triggers, including AUTO and ONE.

`NDIGO6G12_ADC_CHANNEL_COUNT`

The number of analog input channels.

`NDIGO6G12_GATE_COUNT`

The number of gating blocks.

`NDIGO6G12_TDC_CHANNEL_COUNT`

The number of high (TDC0-3) and low (TRG, GATE) resolution TDC input channels.

`NDIGO6G12_BITSTREAM_DATE_LEN`

Bitstream date format: YYYY-MM-DD hh:mm:ss

`NDIGO6G12_CALIBRATION_DATE_LEN`

Calibration date format: YYYY-MM-DD hh:mm

`NDIGO6G12_FLASH_SIG_LEN`

Length of Ndigo6G-12 flash signature

`NDIGO6G12_FIFO_DEPTH`

ADC sample FIFO depth.

It is the maximum recording length in multiples of 5 ns.

`NDIGO6G12_MAX_PRECURSOR`

Maximum for *ndigo6g12_trigger_block::precursor*.

`NDIGO6G12_MAX_MULTISHOT`

Maximum for *ndigo6g12_trigger_block::multi_shot_count*.

### 3.1.2 Trigger and Gating Block Sources

Bitmasks for trigger sources.

Used for *ndigo6g12_trigger_block::sources*, *ndigo6g12_gating_block::sources*, *ndigo6g12_tdc_gating_block::sources*, and *ndigo6g12_tdc_tiger_block::sources*.

***Defines***

`NDIGO6G12_TRIGGER_SOURCE_NONE`

All trigger sources disabled.

`NDIGO6G12_TRIGGER_SOURCE_A0`

`NDIGO6G12_TRIGGER_SOURCE_A1`

`NDIGO6G12_TRIGGER_SOURCE_B0`

`NDIGO6G12_TRIGGER_SOURCE_B1`

`NDIGO6G12_TRIGGER_SOURCE_C0`

`NDIGO6G12_TRIGGER_SOURCE_C1`

`NDIGO6G12_TRIGGER_SOURCE_D0`

`NDIGO6G12_TRIGGER_SOURCE_D1`

`NDIGO6G12_TRIGGER_SOURCE_TDC0`

`NDIGO6G12_TRIGGER_SOURCE_TDC1`

`NDIGO6G12_TRIGGER_SOURCE_TDC2`

`NDIGO6G12_TRIGGER_SOURCE_TDC3`

`NDIGO6G12_TRIGGER_SOURCE_TRG`

`NDIGO6G12_TRIGGER_SOURCE_GATE`

`NDIGO6G12_TRIGGER_SOURCE_AUTO`

`NDIGO6G12_TRIGGER_SOURCE_ONE`

Trigger signal is active each clock cycle.

`NDIGO6G12_TRIGGER_SOURCE_FPGA0`

Deprecated. Alias for *NDIGO6G12_TRIGGER_SOURCE_TRG*.

`NDIGO6G12_TRIGGER_SOURCE_FPGA1`

Deprecated. Alias for *NDIGO6G12_TRIGGER_SOURCE_GATE*.

### 3.1.3 Function return values

Return codes of various functions.

All ERRORS must be positive integers, because the upper byte is used by crono_tools

***Defines***

`CRONO_OK`

`CRONO_WINDRIVER_NOT_FOUND`

`CRONO_DEVICE_NOT_FOUND`

`CRONO_NOT_INITIALIZED`

`CRONO_WRONG_STATE`

`CRONO_INVALID_DEVICE`

`CRONO_BUFFER_ALLOC_FAILED`

`CRONO_TDC_NO_EDGE_FOUND`

`CRONO_INVALID_BUFFER_PARAMETERS`

`CRONO_INVALID_CONFIG_PARAMETERS`

CRONO_WINDOW_CALIBRATION_FAILED

CRONO_HARDWARE_FAILURE

CRONO_INVALID_ADC_MODE

CRONO_SYNCHRONIZATION_FAILED

CRONO_DEVICE_OPEN_FAILED

CRONO_INTERNAL_ERROR

CRONO_CALIBRATION_FAILURE

CRONO_INVALID_ARGUMENTS

CRONO_INSUFFICIENT_DATA

### 3.1.4 PCIe Information

PCIe correctable error flags.

Only relevant when troubleshooting.

*Defines*

CRONO_PCIE_RX_ERROR

CRONO_PCIE_BAD_TLP

CRONO_PCIE_BAD_DLLP

CRONO_PCIE_REPLAY_NUM_ROLLOVER

CRONO_PCIE_REPLAY_TIMER_TIMEOUT

CRONO_PCIE_ADVISORY_NON_FATAL

CRONO_PCIE_CORRECTED_INTERNAL_ERROR

CRONO_PCIE_HEADER_LOG_OVERFLOW

PCIe uncorrectable error flags.

Only relevant when troubleshooting.

***Defines***

`CRONO_PCIE_UNC_UNDEFINED`

`CRONO_PCIE_UNC_DATA_LINK_PROTOCOL_ERROR`

`CRONO_PCIE_UNC_SURPRISE_DOWN_ERROR`

`CRONO_PCIE_UNC_POISONED_TLP`

`CRONO_PCIE_UNC_FLOW_CONTROL_PROTOCOL_ERROR`

`CRONO_PCIE_UNC_COMPLETION_TIMEOUT`

`CRONO_PCIE_UNC_COMPLETER_ABORT`

`CRONO_PCIE_UNC_UNEXPECTED_COMPLETION`

`CRONO_PCIE_UNC_RECEIVER_OVERFLOW_ERROR`

`CRONO_PCIE_UNC_MALFORMED_TLP`

`CRONO_PCIE_UNC_ECRC_ERROR`

`CRONO_PCIE_UNC_UNSUPPORED_REQUEST_ERROR`

## 3.2 Initialization

To use a Ndigo6G-12 board, it first needs to be initialized. This is done by calling *ndigo6g12_init()*. The initialization parameters necessary for *ndigo6g12_init()* are provided in the *ndigo6g12_init_parameters* struct.

The general procedure for initialization is as follows:

1. Load a default set of initialization parameters using *ndigo6g12_get_default_init_parameters*.
2. If necessary, adjust default parameters to your specific needs.
3. Initialize the Ndigo6G-12 board using *ndigo6g12_init()*.
4. Check that the initialization was successful. If so, the return value of *ndigo6g12_init()* is *CRONO_OK*.

Information on the current device will be stored as type *ndigo6g12_device*.

**ndigo6g12_get_default_init_parameters**(init)

Macro that calls *ndigo6g12_get_default_init_parameters_version* with the correct API version.

int **ndigo6g12_get_default_init_parameters_version**(*ndigo6g12_init_parameters* *init, int client_api_version)

Sets up the standard parameters.

Gets a set of default parameters for *ndigo6g12_init()*. This must always be used to initialize the *ndigo6g12_init_parameters* structure.

For convinience, the macro *ndigo6g12_get_default_init_parameters* is provided, which automatically sets the correct `client_api_version`.

Default values:

- *card_index* = 0
- *board_id* = 0
- *buffer_size[0]* = 64 (MiB)
- *buffer_size[1-7]* = 0 (unused)
- *dma_read_delay* = 1000
- *perf_derating* = 0
- *led_flashing_mode* = 1
- *clock_source* = *NDIGO6G12_CLOCK_SOURCE_INTERNAL*
- *application_type* = *NDIGO6G12_APP_TYPE_CURRENT*
- *force_bitstream_update* = `false`
- *partial_bitstream_size* = 0
- *partial_bitstream* = `nullptr`
- *firmware_locations* = `nullptr`

**Parameters**

- **init** – **[in]** Pointer to a structure in which to store the initialization values.
- **client_api_version** – **[in]** *NDIGO6G12_API_VERSION*

**Returns**

See *Function return values*.

int **ndigo6g12_init**(*ndigo6g12_device* *device, *ndigo6g12_init_parameters* *params, const char **error_message)

Open and initialize an Ndigo6G-12 board.

Which Ndigo6G-12 board will be initialized is determined by *ndigo6g12_init_parameters::card_index*.

**Parameters**

- **device** – **[out]** Pointer to the device struct.

- **params** – **[in]** Pointer to the structure that contains the initialization parameters.
- **error_message** – **[out]** Location in which to store the error message as plain text.

**Returns**

See *Function return values*.

int **ndigo6g12_close**(*ndigo6g12_device* *device)

Finalize the driver for this device.

**Parameters**

**device** – **[in]** Pointer to the device that should be finalized.

**Returns**

See *Function return values*.

struct **ndigo6g12_device**

Contains information of the Ndigo6G-12 device in use.

***Public Members***

bool **is_valid**

void ***ndigo6g12**

struct **ndigo6g12_init_parameters**

Struct for the initialization of the Ndigo6G-12.

This structure MUST be completely initialized.

***Public Members***

int **version**

The version number.

It is increased when the definition of the structure is changed. The increment can be larger than 1 to match driver version numbers or similar. Set to 0 for all versions up to first release.

Must be set to *NDIGO6G12_API_VERSION*.

int **card_index**

The index in the list of Ndigo6G-12 boards that should be initialized.

There might be multiple boards installed in the system that are handled by this driver as reported by *ndigo6g12_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 the list of 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 Ndigo6G-12 boards are used, this number can be set to *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.

int64_t **buffer_size**[8]

The minimum size of the DMA buffer.

If set to 0, the default size of 64 MiBytes is used. For the Ndigo6G-12 only the first entry is used.

int **dma_read_delay**

The update delay of the writing pointer after a packet has been send over PCIe.

Default is 1000. Do not change.

int **perf_derating**

Default 0, corresponding to 1.6, 3.2, or 6.4 Gsps (depending on `application_type`).

For internal use only. Do not change.

int **led_flashing_mode**

Controls the LED flashing mode.

Define what LEDs do during initialization:

- 0: LEDs are off
- 1: LEDs light up once

int **clock_source**

Defines which clock source is used (internal, SMA, AUX2).

Must be one of the following:

**NDIGO6G12_CLOCK_SOURCE_INTERNAL**

Device is using the internal 10 MHz clock.

**NDIGO6G12_CLOCK_SOURCE_SMA**

Use an external 10 MHz clock as reference. The input is the SMA socket located on the board.

**NDIGO6G12_CLOCK_SOURCE_AUX2**

Use an external 10 MHz clock as reference. The input is the TRG LEMO connector located on the slot bracket.

uint32_t **application_type**

Select the application type.

Note that *ndigo6g12_configuration::adc_mode* must match the application type chosen here.

Must be one of the following:

**NDIGO6G12_APP_TYPE_AVRG**

Averaging mode at 6.4 Gsps.

For more information, see Section 2.6.

**NDIGO6G12_APP_TYPE_4CH**

Four ADC channels at 1.6 Gsps.

**NDIGO6G12_APP_TYPE_2CH**

Two ADC channels at 3.2 Gsps.

**NDIGO6G12_APP_TYPE_1CH**

One ADC channel at 6.4 Gsps.

**NDIGO6G12_APP_TYPE_CURRENT**

Use currently installed application type.

crono_bool_t **force_bitstream_update**

Force a bitstream update that configures the FPGA.

During the initialization of the board, a bitstream configures the FPGA of the Ndigo6G-12. This is only done if during the initialization of the Ndigo6G-12, `application_type` is different from the `application_type` that the Ndigo6G-12 is currently configured in. That is, the FPGA is only reconfigured, if `application_type` changes.

By setting `force_bitstream_update` to `true`, one can force a reconfiguration of the FPGA.

int **partial_bitstream_size**

Size of *partial_bitstream*.

Reserved for future expandability.

uint32_t ***partial_bitstream**

Pointer to a buffer with partial bitstream data.

Can be `nullptr` if *application_type* matches `application_type` of currently installed firmware.

Reserved for future expandability.

const char ***firmware_locations**

Location where firmware is installed.

Pointer to a list of paths (separated by `;`) Can be `nullptr` if *application_type* matches `application_type` of currently installed firmware.

## 3.3 Status information

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

int **ndigo6g12_get_driver_revision**()

Get the driver version in integer format.

**Returns**
The driver version in the same format as *ndigo6g12_static_info::driver_revision*.

const char ***ndigo6g12_get_driver_revision_str**()

Get the driver version in string format.

**Returns**
The Driver version including SVN build revision as a string with format x.y.z.svn.

int **ndigo6g12_count_devices**(int *error_code, const char **error_message)

Get the number of Ndigo6G-12 boards that are installed in the system.

**Parameters**

- **error_code** – **[out]** Pointer to an integer in which to store the *error code*.
- **error_message** – **[out]** Location in which to store the error message as plain text.

**Returns**
The number.

int **ndigo6g12_get_static_info**(*ndigo6g12_device* *device, *ndigo6g12_static_info* *static_info)

Get the static information.

The static information does not change after the device initialization.

**Parameters**

- **device** – **[in]** Pointer to the device from which to get the information.
- **static_info** – **[out]** Pointer to a structure in which to store the information.

**Returns**
See *Function return values*.

int **ndigo6g12_get_param_info**(*ndigo6g12_device* *device, *ndigo6g12_param_info* *param_info)

Get parametric information.

The parametric information may change due to the configuration.

**Parameters**

- **device** – **[in]** Pointer to the device from which to get the information.
- **param_info** – **[out]** Pointer to a structure in which to store the information.

**Returns**
See *Function return values*.

int **ndigo6g12_get_fast_info**(*ndigo6g12_device* *device, *ndigo6g12_fast_info* *fast_info)

Get fast status information.

The information can be retrieved within a few microseconds.

**Parameters**

- **`device`** – **[in]** Pointer to the device from which to get the information.
- **`fast_info`** – **[out]** Pointer to a structure in which to store the information.

**Returns**

See *Function return values*.

int **`ndigo6g12_get_pcie_info`**(*ndigo6g12_device* *device, *crono_pcie_info* *pcie_info)

Reads the PCIe info like correctable and uncorrectable errors.

**Parameters**

- **`device`** – **[in]** Pointer to the device.
- **`pcie_info`** – **[out]** Pointer to the structure in which to store the information.

**Returns**

See *Function return values*.

struct **`ndigo6g12_param_info`**

Contains configuration changes.

Structure filled by *ndigo6g12_get_param_info()*. This structure contains information that may change indirectly due to configuration changes.

***Public Members***

double **`bandwidth`**

Bandwidth.

4.5 or 6.5 GHz depending on *ndigo6g12_configuration::extended_bandwidth*.

int **`resolution`**

ADC sample resolution.

Always 12 bit.

double **`sample_rate`**

Actual ADC sample rate of currently sampled data.

Depending on *ndigo6g12_configuration::adc_mode*, that is, `sample_rate` = 6.4 GHz / *channels*.

double **`sample_period`**

The period that one sample in the data represents in picoseconds.

double **`tdc_period`**

The period that one TDC bin in the data represents in picoseconds.

double **`packet_ts_period`**

The period that one tick of the packet timestamp represents in picoseconds.

uint64_t **tdc_packet_timestamp_offset**

The TDC packet timestamp offset.

Since TDC packets carry the timestamp of the end of the packet, to calculate the start, `tdc_packet_timestamp_offset` has to be subtracted.

uint32_t **tdc_rollover_period**

Time span of one TDC timestamp rollover period in units of the TDC binsize.

All TDC hits within this period are written to one crono_packet.

double **adc_sample_delay**

The delay of the ADC samples relative to TDC timestamps in picoseconds.

Note: For driver release 2.2.0 with firmware 1.25086, this value is bugged.

int **board_id**

The ID the board uses to identify itself in the output data stream.

Takes values 0 to 255.

int **channels**

Number of ADC channels in the current mode.

See *ndigo6g12_configuration::adc_mode*.

int **channel_mask**

Mask with a set bit for each enabled input channel.

int **tdc_channels**

Number of TDC channels in the current mode.

int64_t **total_buffer**

The total amount of the DMA buffer in bytes.

int **samples_per_clock**

The number of samples in one clock cycle in the current mode.

struct **ndigo6g12_static_info**

Structure contains static information.

This structure contains information about the board that does not change during run time. It is provided by *ndigo6g12_get_static_info()*.

*Public Members*

char **bitstream_date**[NDIGO6G12_BITSTREAM_DATE_LEN]

Bitstream creation date.

DIN EN ISO 8601 string YYYY-MM-DD HH:DD:SS describing the time when the bitstream was created.

int **board_configuration**

Describes the schematic configuration of the board.

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

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.

- 0: Experimental version of the first board. Labeled "Rev. 1".
- 2: First commercial version. Labeled "Rev. 2"

int **board_serial**

The board's serial number.

With year and running number in 8.24 format (yy.nnn; 8 bits are used to encode the year, 24 bits to encode the number).

The number is identical to the one printed on the silvery sticker on the board.

char **calibration_date**[NDIGO6G12_CALIBRATION_DATE_LEN]

Calibration date.

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

int **chip_id**

16-bit factory ID of the ADC chip.

This is the chipID as read from the 16-bit ADC chip-ID register.

crono_bool_t **dc_coupled**

Shows if the inputs are DC-coupled.

Default is `false`, that is, AC-coupled.

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.

A change in the first digit generally requires a recompilation of user applications. Change in the second digit denote significant improvements or changes that don't break compatibility and the third digit changes with minor bugfixes and the like (see https://semver.org/).

int **driver_build_revision**

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

crono_bool_t **flash_valid**

Calibration data read from flash is valid.

If not `false`, the driver found valid calibration data in the flash on the board and is using it.

int **fw_revision**

Revision number of the FPGA configuration.

int **fw_type**

Type of firmware, always 5 -> Ndigo6G-12.

int **pcb_serial**

Trenz serial number.

int **svn_revision**

Subversion revision ID of the FPGA configuration.

A number to track builds of the firmware in more detail than the firmware revision. It changes with every change in the firmware, even if there is no visible effect for the user. The subversion revision number can be read from a register.

int **application_type**

Shows the initialized mode.

See NDIGO6G12_APP_TYPE_* constants.

char **config_flash_signature_primary**[NDIGO6G12_FLASH_SIG_LEN]

Shows the signature of the primary flash.

char **config_flash_signature_secondary**[NDIGO6G12_FLASH_SIG_LEN]

Shows the signature of the secondary flash.

double **auto_trigger_ref_clock**

Auto trigger clock frequency.

The clock frequency of the auto trigger in Hz used for the calculations of *ndigo6g12_configuration::auto_trigger_period*.

Fixed at 200 MHz.

struct **ndigo6g12_fast_info**

Contains fast dynamic information.

This structure is filled by *ndigo6g12_get_fast_info()*. This information can be obtained within a few microseconds.

### Public Members

int **state**

The current state of the device.

Is one of the following:

**NDIGO6G12_DEVICE_STATE_INITIALIZED**

Device is initialized but not yet configured for data capture.

**NDIGO6G12_DEVICE_STATE_CONFIGURED**

Device is ready for data capture.

**NDIGO6G12_DEVICE_STATE_CAPTURING**

Device has started data capture.

int **fan_speed**

Speed of the FPGA fan in rounds per minute.

Reports 0 if no fan is present.

double **fpga_temperature**

Temperature of the FPGA in °C.

double **fpga_vccint**

Internal Voltage of the FPGA in V. Useful debugging information.

double **fpga_vccaux**

Auxillary Voltage of the FPGA in V. Useful debugging information.

double **fpga_vccbram**

BRAM Voltage of the FPGA in V. Useful debugging information.

double **mgt_0v9**

Shows measured voltage for the mgt_0v9 power supply in V. Useful debugging information.

double **mgt_1v2**

Shows measured Voltage for the mgt_1v2 power supply in V. Useful debugging information.

double **adc_2v5**

Shows measured voltage for the 2v5 power supply in V. Useful debugging information.

double **clk_3v3**

Shows measured voltage for the clk_3v3 power supply in V. Useful debugging information.

double **adc_3v3**

Shows measured voltage for the adc_3v3 power supply in V. Useful debugging information.

double **pcie_3v3**

Shows measured voltage for the pcie_3v3 power supply in V. Useful debugging information.

double **opamp_5v2**

Shows measured voltage for the opamp_5v2 power supply in V. Useful debugging information.

double **temp4633_1**

Shows temperature of voltage regulartor U3_1 in °C.

double **temp4633_2**

Shows temperature of voltage regulator U3_2 in °C.

double **temp4644**

Shows temperature of voltage regulator U4 in °C.

double **tdc1_temp**

Temperature of the TDC-chip in °C.

double **ev12_cmiref**

Shows voltage for differential ADC input common mode voltage in V.

Measured or calibration target depending on board revision and assembly variant.

double **ev12_temp**

Temperature of the ADC in °C.

int **alerts**

Alert bits from temperature sensor and the system monitor.

Bit 0 is set if the TDC temperature exceeds 140°C. In this case the TDC shut down and the device needs to be reinitialized.

Is one of the following:

**NDIGO6G12_ALERT_FPGA_TEMPERATURE**

FPGA temperature alert (> 70°C)

**NDIGO6G12_ALERT_VCCINT**

Internal FPGA voltage out of range (< 0.83 V or > 0.88 V).

**NDIGO6G12_ALERT_VCCAUX**

FPGA auxiliary voltage out of range (< 1.75 V or > 1.89 V).

**NDIGO6G12_ALERT_FPGA_TEMPERATURE_CRITICAL**

FPGA temperature critical (> 80°C)

**NDIGO6G12_ALERT_THS_TEMPERATURE_CRITICAL**

THS temperature critical (> 140°C)

int **pcie_link_width**

Number of PCIe lanes the card uses.

Should always be 8 for the Ndigo6G-12.

int **pcie_link_speed**

Data rate of the PCIe card.

Should always be 3 for the Ndigo6G-12.

int **pcie_max_payload**

Maximum size for a single PCIe transaction in bytes.

Depends on the system configuration.

crono_bool_t **adc_data_pll_locked**

ADC data clock is PLL locked.

crono_bool_t **adc_data_pll_lost_lock**

ADC data clock PLL lost lock (Sticky Bit).

int **adc_lanes_synced**

Shows the synced ADC lanes.

Each bit corresponds to one lane. Useful debugging information.

int **adc_lanes_lost_sync**

Shows the ADC lanes that lost sync.

Each bit corresponds to one lane. Useful debugging information.

int **adc_lanes_fifo_empty**

Shows which ADC lanes have an empty FIFO.

Each bit corresponds to one lane. Useful debugging information.

int **adc_lanes_fifo_full**

Shows which ADC lanes have a full FIFO.

Each bit corresponds to one lane. Useful debugging information.

int **adc_lanes_running**

Shows which ADC lanes are running.

Each bit corresponds to one lane. Useful debugging information.

int **adc_lanes_sync_timeout**

Shows which ADC lanes were unable to sync before a timeout.

Each bit corresponds to one lane. Useful debugging information.

int **adc_sync_retry_count**

The number of ADC lane synchronization retries.

Default is set to 0. Useful debugging information.

int **adc_sync_strobe_retry_count**

The number of ADC strobe synchronization retries.

Default is set to 0. Useful debugging information.

int **adc_sync_delay_count**

16 Bit number showing when the last ADC lane synchronization was achieved.

Useful debugging information.

crono_bool_t **adc_mgt_power_good**

Shows if the supplied mgt power is sufficient.

Useful debugging information.

crono_bool_t **lmk_pll1_locked**

Shows if lmk_pll1 is locked. Useful debugging information.

crono_bool_t **lmk_pll2_locked**

Shows if lmk_pll2 is locked. Useful debugging information.

crono_bool_t **lmk_lost_lock**

Shows if lmk lost lock. Useful debugging information.

int **lmk_lock_wait_count**

Wait count of the lmk. Useful debugging information.

int **lmk_ctrl_vcxo**

Usefull for hardware debugging.

crono_bool_t **lmx_locked**

lmx locked. Useful debugging information.

crono_bool_t **lmx_lost_lock**

lmx lost lock. Useful debugging information.

int **lmx_lock_wait_count**

lmx lock wait count. Useful debugging information.

struct **crono_pcie_info**

Structure containing PCIe information.

*Public Members*

uint32_t **pwr_mgmt**

Organizes power supply of PCIe lanes.

uint32_t **link_width**

Number of PCIe lanes that the card uses.

Should be 1, 2, or 4 for Ndigo5G and 1, 2, 4, or 8 for the Ndigo6G-12. Ideally, should be the respective maximum.

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. 0, if no error is present, otherwise one of *CRONO_PCIE_** .

uint32_t **uncorrectable_error_status**

Uncorrectable error status flags, directly from the PCIe config register.

Useful for debugging PCIe problems. 0, if no error is present, otherwise one of *CRONO_PCIE_UNC_** .

uint32_t **reserved**

For future extension.

## 3.4 Configuration

The Ndigo6G-12 board is configured with a configuration structure (`ndigo6g12_configuration`).

The user should first obtain a standard set of configuration parameters using `ndigo6g12_get_default_configuration()`, then modify only the necessary parameters to their specific needs.

The configuration itself is done by calling `ndigo6g12_configure()`.

int **`ndigo6g12_get_default_configuration`**(*ndigo6g12_device* *device, *ndigo6g12_configuration* *config)

Copies the default configuration to the specified config pointer.

Default values of *ndigo6g12_configuration*:

- *adc_mode* =
  - *NDIGO6G12_ADC_MODE_A* (if *application_type* = *NDIGO6G12_APP_TYPE_1CH*)
  - *NDIGO6G12_ADC_MODE_AD* (if *application_type* = *NDIGO6G12_APP_TYPE_2CH*)
  - *NDIGO6G12_ADC_MODE_ABCD* (if *application_type* = *NDIGO6G12_APP_TYPE_4CH*)
  - *NDIGO6G12_ADC_MODE_A* (if *application_type* = *NDIGO6G12_APP_TYPE_AVRG*)
- *adc_cal_set* = 3
- *analog_offsets[i]* = 0
- *tdc_trigger_offsets[i]* = *NDIGO6G12_DC_OFFSET_N_NIM*
- *trigger[i]*:
  - *edge* = `true`
  - *rising* = `false`
  - *threshold* = 512
- *trigger_block[i]*:
  - *enabled* = `false`
  - *retrigger* = `false`
  - *multi_shot_count* = 1
  - *precursor* = 0
  - *length* = 16
  - *sources* = *NDIGO6G12_TRIGGER_SOURCE_0*
  - *gates* = *NDIGO6G12_TRIGGER_GATE_NONE*
  - *minimum_free_packets* = 0
- *gating_block[i]*:
  - *negate* = `false`
  - *retrigger* = `false`
  - *start* = 0

    - *stop* = 1000
    - *sources* = *NDIGO6G12_TRIGGER_SOURCE_0*
- *tdc_configuration*:
    - *channel[i]*:
        * *enable* = `false`
        * *gating_block*:
            - *enable* = `false`
            - *negate* = `false`
            - *retrigger* = `false`
            - *retrigger* = *NDIGO6G12_TRIGGER_SOURCE_AUTO*
            - *start* = 0
            - *stop* = 1000
            - *sources* = *NDIGO6G12_TRIGGER_SOURCE_0*
        * *tiger_block*:
            - *mode* = *NDIGO6G12_TIGER_OFF*
            - *negate* = `true`
            - *retrigger* = `false`
            - *retrigger* = *NDIGO6G12_TRIGGER_SOURCE_AUTO*
            - *start* = 0
            - *stop* = 1
            - *sources* = *NDIGO6G12_TRIGGER_SOURCE_0*
    - *skip_alignment* = `false`
    - *alignment_mode* = `false`
    - *alignment_pin_high_z* = `false`
    - *alignment_pin_invert* = `false`
    - *alignment_phase_steps* = 6
    - *send_empty_packets* = `false`
- *auto_trigger_period* = 200000
- *auto_trigger_random_exponent* = 0
- *output_mode* =
    - *NDIGO6G12_OUTPUT_MODE_SIGNED32* (if *application_type* = *NDIGO6G12_APP_TYPE_AVRG*)
    - *NDIGO6G12_OUTPUT_MODE_SIGNED16* (otherwise)
- *extended_bandwidth* = `false`

- *ramp_test_mode* = `false`

**Parameters**

- **device** – **[in]** Pointer to the device from which to get the information.
- **config** – **[out]** Pointer to a structure in which to store the configuration values.

**Returns**

See *Function return values*.

int **ndigo6g12_configure**(*ndigo6g12_device* *device, *ndigo6g12_configuration* *config)

Configures the Ndigo6G-12 device.

The config information is copied such that it can be changed after the call to *ndigo6g12_configure*.

**Parameters**

- **device** – **[in]** Pointer to the device from which to get the information.
- **config** – **[out]** Pointer to the configuration structure.

**Returns**

See *Function return values*.

struct **ndigo6g12_configuration**

Structure that contains the configuration values for the Ndigo6G-12.

This structure contains the configuration information. It is used in conjunction with *ndigo6g12_get_default_configuration()* and *ndigo6g12_configure()*.

***Public Members***

int **adc_mode**

Configure ADC mode.

The chosen ADC mode has to be supported by the current NDIGO6G12_APP_TYPE.

For example, if `NDIGO6G12_APP_TYPE_1CH` is used, one *cannot* choose, e.g., `adc_mode = NDIGO6G12_ADC_MODE_AA`, but one has to either choose `NDIGO6G12_ADC_MODE_A` or `NDIGO6G12_ADC_MODE_D`.

Default value depends on *ndigo6g12_init_parameters::application_type*.

- *NDIGO6G12_APP_TYPE_4CH*: *NDIGO6G12_ADC_MODE_A*
- *NDIGO6G12_APP_TYPE_2CH*: *NDIGO6G12_ADC_MODE_AD*
- *NDIGO6G12_APP_TYPE_1CH*: *NDIGO6G12_ADC_MODE_ABCD*

For more information, see Section 2.1.

Must be one of the following:

**NDIGO6G12_ADC_MODE_ABCD**

4-channel mode at 1600 Msps sample rate

**NDIGO6G12_ADC_MODE_AADD**

4-channel mode at 1600 Msps sample rate

**NDIGO6G12_ADC_MODE_AAAA**

4-channel mode at 1600 Msps sample rate

**NDIGO6G12_ADC_MODE_DDDD**

4-channel mode at 1600 Msps sample rate

**NDIGO6G12_ADC_MODE_AD**

2-channel mode at 3200 Msps sample rate

**NDIGO6G12_ADC_MODE_AA**

2-channel mode at 3200 Msps sample rate

**NDIGO6G12_ADC_MODE_DD**

2-channel mode at 3200 Msps sample rate

**NDIGO6G12_ADC_MODE_A**

1-channel mode at 6400 Msps sample rate

**NDIGO6G12_ADC_MODE_D**

1-channel mode at 6400 Msps sample rate

int **adc_cal_set**

Select ADC calibration set.

Default is 3. Do not change.

double **analog_offsets**[NDIGO6G12_ADC_CHANNEL_COUNT]

Set the offsets of the ADC inputs in V.

The indices 0 to 3 of the array correspond to ADC channels A to D.

Must be between $\pm$ 0.5 V.

Defaults are 0 V for each ADC channel.

double **tdc_trigger_offsets**[NDIGO6G12_TDC_CHANNEL_COUNT]

Set DAC for trigger threshold of the TDC inputs in V.

Channel assignment:

- 0 to 3: high-resolution TDC, inputs E to H
- 4 and 5: inputs TRG and GATE

Set to a value between −1.32 V and +2.0 V.

This should be close to 50% of the height of your pulses on the inputs. Examples for various signaling standards are defined below. The inputs are AC coupled. This means that for pulse inputs the absolute voltage is not important. Only the relative pulse amplitude causes the input circuits to switch. `tdc_trigger_offset` for an input 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.

Defaults are *NDIGO6G12_DC_OFFSET_N_NIM* for each TDC channel.

Defines for various signal standards:

`NDIGO6G12_DC_OFFSET_P_NIM`

`NDIGO6G12_DC_OFFSET_P_CMOS`

`NDIGO6G12_DC_OFFSET_P_LVCMOS_33`

`NDIGO6G12_DC_OFFSET_P_LVCMOS_25`

`NDIGO6G12_DC_OFFSET_P_LVCMOS_18`

`NDIGO6G12_DC_OFFSET_P_TTL`

`NDIGO6G12_DC_OFFSET_P_LVTTL_33`

`NDIGO6G12_DC_OFFSET_P_LVTTL_25`

`NDIGO6G12_DC_OFFSET_P_SSTL_3`

`NDIGO6G12_DC_OFFSET_P_SSTL_2`

`NDIGO6G12_DC_OFFSET_N_NIM`

`NDIGO6G12_DC_OFFSET_N_CMOS`

`NDIGO6G12_DC_OFFSET_N_LVCMOS_33`

`NDIGO6G12_DC_OFFSET_N_LVCMOS_25`

`NDIGO6G12_DC_OFFSET_N_LVCMOS_18`

`NDIGO6G12_DC_OFFSET_N_TTL`

`NDIGO6G12_DC_OFFSET_N_LVTTL_33`

`NDIGO6G12_DC_OFFSET_N_LVTTL_25`

NDIGO6G12_DC_OFFSET_N_SSTL_3

NDIGO6G12_DC_OFFSET_N_SSTL_2

*ndigo6g12_trigger* **trigger**[NDIGO6G12_TRIGGER_COUNT]

Configuration of the external trigger sources.

The entries in the array correspond to the following defines.

*ndigo6g12_trigger::threshold* is ignored for index *NDIGO6G12_TRIGGER_TDC0* and above.

*ndigo6g12_trigger::edge* and *ndigo6g12_trigger::rising* are ignored for indeces *NDIGO6G12_TRIGGER_AUTO* and *NDIGO6G12_TRIGGER_ONE*.

NDIGO6G12_TRIGGER_A0

NDIGO6G12_TRIGGER_A1

NDIGO6G12_TRIGGER_B0

NDIGO6G12_TRIGGER_B1

NDIGO6G12_TRIGGER_C0

NDIGO6G12_TRIGGER_C1

NDIGO6G12_TRIGGER_D0

NDIGO6G12_TRIGGER_D1

NDIGO6G12_TRIGGER_TDC0

NDIGO6G12_TRIGGER_TDC1

NDIGO6G12_TRIGGER_TDC2

NDIGO6G12_TRIGGER_TDC3

NDIGO6G12_TRIGGER_TRG

NDIGO6G12_TRIGGER_GATE

NDIGO6G12_TRIGGER_AUTO

**NDIGO6G12_TRIGGER_ONE**

**NDIGO6G12_TRIGGER_FPGA0**

Deprecated. Alias for *NDIGO6G12_TRIGGER_TRG*.

**NDIGO6G12_TRIGGER_FPGA1**

Deprecated. Alias for *NDIGO6G12_TRIGGER_GATE*.

*ndigo6g12_trigger_block* **trigger_block**[NDIGO6G12_ADC_CHANNEL_COUNT]

Trigger settings of ADC inputs.

The number of input channels depends on ADC mode.

*ndigo6g12_gating_block* **gating_block**[NDIGO6G12_GATE_COUNT]

Configuration of gating blocks.

Gating blocks are used to filter trigger.

*ndigo6g12_tdc_configuration* **tdc_configuration**

Configuration of TDC channels.

*ndigo6g12_averager_configuration* **average_configuration**

Configuration of the Averager.

int **auto_trigger_period**

Component to create a trigger either periodically or randomly.

To be exact, there are two parameters $M$ = *auto_trigger_period* and $N$ = *auto_trigger_random_exponent* that result in a distance between triggers of $T = M + [1 \ldots 2^N] - 1$ clock cycles, where $6 \le M < 2^{32}$ and $0 \le N < 32$.

There is no enable or reset as the usage of this trigger can be configured in the channels. Each clock cycle is 5 ns.

Default is 200000, corresponding to a 1 kHz auto trigger.

int **auto_trigger_random_exponent**

Component to create a trigger either periodically or randomly.

See *auto_trigger_period*.

Default is 0.

int **output_mode**

Output mode of the ADC data.

Default value depends on *ndigo6g12_init_parameters::application_type*.

- *NDIGO6G12_APP_TYPE_AVRG*: *NDIGO6G12_OUTPUT_MODE_SIGNED32*
- otherwise: *NDIGO6G12_OUTPUT_MODE_SIGNED16*.

Must be one of the following:

**NDIGO6G12_OUTPUT_MODE_RAW**

Return the native range (0 to 4095) augmented by two ADC control bits per sample.

Not supported for user applications.

**NDIGO6G12_OUTPUT_MODE_SIGNED16**

Return a signed16 integer.

The range is -32768 to 32767.

**NDIGO6G12_OUTPUT_MODE_SIGNED32**

Output in signed32 integer format.

Must be used in (and only in) averaging mode. The range is $-2^{31}$ to $2^{31} - 1$.

**NDIGO6G12_OUTPUT_MODE_RAW_NO_CB**

Return the native range (0 to 4095).

For more information, see Section 4.3.

crono_bool_t **extended_bandwidth**

Extended bandwidth.

If `true`, the input bandwidth is 6.5 GHz instead of the default 4.5 GHz.

Since the extended input bandwidth of the ADC influences the total bandwidth of the Ndigo6G-12 board only in a minimal manner, we recommend using the non-extended input bandwidth of 4.5 GHz. This ensures the best signal-to-noise ratio.

Default is `false`.

crono_bool_t **ramp_test_mode**

Default is `false`. Do not change.

crono_bool_t **sample_averaging**

Calculate sample average for multi-sampling modes *AAAA*, *DDDD*, *AADD*, *AA*, and *DD*.

Manipulate the output in multi-sampling modes.

- `true`: Average all samples and combine them to a single output.
- `false`: Output all samples in their own package.

For more information, see *Multiple Sampling Modes* in Section 2.1.

struct **ndigo6g12_trigger**

Structure that contains trigger settings.

### Public Members

short **threshold**

Threshold controlling when the ADC channel is active.

Sets the threshold for the trigger block within the range of the ADC data. The range depends on *ndigo6g12_configuration::output_mode*:

- *NDIGO6G12_OUTPUT_MODE_RAW* : 0 to 4095
- *NDIGO6G12_OUTPUT_MODE_SIGNED16* and *NDIGO6G12_OUTPUT_MODE_SIGNED32* : -32768 to 32767

For trigger indices NDIGO6G12_TRIGGER_TDC to *NDIGO6G12_TRIGGER_ONE* the threshold is ignored.

For the TDC channels, the trigger threshold is controlled by *ndigo6g12_configuration::tdc_trigger_offsets*.

> **Note**
>
> `NDIGO6G12_OUTPUT_MODE_SIGNED32` is only used for `NDIGO6G12_APP_TYPE_AVRG`.

crono_bool_t **edge**

Enables edge-trigger functionality.

For trigger indices *NDIGO6G12_TRIGGER_AUTO* and *NDIGO6G12_TRIGGER_ONE* this is ignored.

- `false`: Use a level trigger. The level trigger triggers as long as the signal is above or below (depending on `rising`) the set threshold. Followingly, the trigger gives the sign of the signal in reference to the threshold.
- `true`: Use an edge trigger. The edge trigger triggers as soon as its set threshold is crossed by the signal. Thus, the roots in reference to the threshold are recorded.

Default is `true`.

crono_bool_t **rising**

Sets rising-edge trigger functionality.

For trigger indices *NDIGO6G12_TRIGGER_AUTO* and *NDIGO6G12_TRIGGER_ONE*, this is ignored.

- If `edge` is `true` (i.e., an edge trigger is used):
  - `false`: Trigger when the signal crosses from above to below the threshold.
  - `true`: Trigger when the signal crosses from below to above the threshold.
- If `edge` is `false` (i.e., a level trigger is used):
  - `false`: Triggers the part of the signal below the threshold.
  - `true`: Triggers the part of the signal above the threshold.

Default is `false`.

struct **ndigo6g12_trigger_block**

Configuration of the trigger block.

### *Public Members*

crono_bool_t **enabled**

Activates triggers on this channel.

crono_bool_t **retrigger**

Enable retrigger functionality.

If a new trigger condition occurs while the postcursor is acquired (i.e., within the time frame controlled by *length*), the packet is extended by starting a new postcursor. Otherwise the new trigger is ignored and the packet ends after the postcursor of the first trigger.

int **multi_shot_count**

Number of packets created in single-shot mode (i.e., *ndigo6g12_single_shot()* was called) before packet generation stops.

This value is ignored if `enabled` is `true`.

Maximum is *NDIGO6G12_MAX_MULTISHOT*.

Note: Up to firmware revision `1.24120`, this feature is bugged in 4-channel mode while `multi_shot_count > 1`.

int **precursor**

Precursor in multiples of 5 ns.

The amount of data preceding a trigger that is captured. The maximum is *NDIGO6G12_MAX_PRECURSOR*.

int **length**

Length of the postcursor in multiples of 5 ns.

The total amount of data that is recorded in addition to the trigger window is controlled by `length` and `precursor`. `precursor` determines the amount of data before the trigger window, `length` the amount of data after the trigger condition was false the first time.

In *edge-trigger mode* , the *trigger window* is always `1` (i.e., 5 ns long). Otherwise, (level-trigger mode) the trigger window is as long as the trigger condition was fulfilled.

The maximum value is *NDIGO6G12_FIFO_DEPTH* minus *ndigo6g12_trigger_block::precursor* minus *trigger window*.

int **sources**

A *bit mask* with a bit set for all trigger sources that can trigger this channel.

Default *NDIGO6G12_TRIGGER_SOURCE_0* (*NDIGO6G12_TRIGGER_SOURCE_A0* for ADC channel A, *NDIGO6G12_TRIGGER_SOURCE_B0* for ADC channel B, etc).

int **gates**

A bit mask with a bit set for all trigger gates.

Mask which selects the gates that have to be open for the trigger block to use.

Default *NDIGO6G12_TRIGGER_GATE_NONE*.

The following defines can be used to create the bit mask:

**NDIGO6G12_TRIGGER_GATE_NONE**

**NDIGO6G12_TRIGGER_GATE_0**

**NDIGO6G12_TRIGGER_GATE_1**

**NDIGO6G12_TRIGGER_GATE_2**

**NDIGO6G12_TRIGGER_GATE_3**

double **minimum_free_packets**

Number of packets that fit into the FIFO.

This parameter sets how many packets are supposed to fit into the on-board FIFO before a new packet is recorded after the FIFO was full, i.e., a certain amount of free space in the FIFO is demanded before a new packet is written after the FIFO was full. As a measure for the packet length, the recording window as defined by *precursor* and *length* is used.

The on-board algorithm checks the free FIFO space only in case the FIFO is full. Therefore, if this number is `1.0` or more, at least every second packet in the host buffer is guaranteed to have the full length set by the *precursor* and *length*. In many cases smaller values will also result in full length packets. But below a certain value multiple packets that are cut off at the end will show up.

Default is 0.

struct **ndigo6g12_gating_block**

Contains settings of the gating block.

After a signal at one of the `sources` is detected, a timer starts running. Once the timer reaches the value specified by `start`, a gate is opened (or closed, depending on `negate`) until the timer reaches the time specified by `stop`.

What happens in the event that another signal before `stop` is detected is controlled by `retrigger`. See also Section 2.4.

***Public Members***

crono_bool_t **negate**

Invert output polarity.

If `false` (`true`) the gate is opened (closed) inbetween the times specified by `start` and `stop`.

Default is `false`.

crono_bool_t **`retrigger`**

Enable retriggering.

If enabled and a second trigger event is detected before the timer reaches `stop`, the timer is restarted. Otherwise signals at the input `sources` are ignored until `stop` is reached.

Default is `false`.

int **`start`**

The time from the first input signal seen in the idle state until the gating output is set.

In multiples of 5 ns. 0 ≤ `start` < $2^{16}$, while `start` ≤ `stop`.

Default is 0.

int **`stop`**

The number of samples from leaving the idle state until the gating output is reset.

In multiples of 5 ns. 0 ≤ `stop` < $2^{16}$, while `stop` ≥ `start`.

Default is 1000.

int **`sources`**

*Bit mask* with a bit set for all trigger sources that can trigger this channel.

Default *NDIGO6G12_TRIGGER_SOURCE_0* (*NDIGO6G12_TRIGGER_SOURCE_A0* for ADC channel A, *NDIGO6G12_TRIGGER_SOURCE_B0* for ADC channel B, etc).

struct **`ndigo6g12_tdc_configuration`**

Contains configuration information of the TDC channels.

### Public Members

*ndigo6g12_tdc_channel* **`channel`**[NDIGO6G12_TDC_CHANNEL_COUNT]

Configure polarity, type and threshold for the TDC channels.

crono_bool_t **`skip_alignment`**

Configure THS788 calibration.

- `true`: Skip THS788 calibration.
- `false`: Do THS788 calibration (default).

Default is `false`.

crono_bool_t **`alignment_mode`**

Align TDC channels.

Default is `false`.

crono_bool_t **alignment_pin_high_z**

Default is `false`.

crono_bool_t **alignment_pin_invert**

Default is `false`.

int **alignment_phase_steps**

Default is 6.

crono_bool_t **send_empty_packets**

Default is `false`.

struct **ndigo6g12_averager_configuration**

Contains averaging settings.

### *Public Members*

int **iterations**

Set the number of trigger events that are averaged.

Must be 0 if no averaging application is installed on the Ndigo6G-12 (see *ndigo6g12_init_parameters::application_type*).

Default is 0.

crono_bool_t **stop_on_overflow**

Stops averaging before an overflow can happen.

Stops the averaging once *averaging_value* ≥ *max_averaging_value* − *max_ADC_value* or *averaging_value* ≤ *min_averaging_value* − *min_ADC_value* to prevent overflow.

- *max(min)_averaging_value* is 2097151 (-2097152)
- *max(min)_ADC_value* is 32768 (-32767)

Default is `false`.

crono_bool_t **stop_manual**

Stops the averaging manually.

Software stop for averaging. If an averaging iteration has already started it is finished before the averaging will stop.

Default is `false`.

crono_bool_t **use_saturation**

Determines if saturation arithmetic is used by the averager.

- `true`: Instead of *averaging_value* over(under)flowing once *max(min)_averaging_value* is reached, the maximum (minimum) value is kept.

- `false`: Once *averaging_value* reaches *max(min)_averaging_value*, *averaging_value* will over(under)flow and wrap around.

See *stop_on_overflow* for the values of *averaging_value* and *max(min)_averaging_value*.

Default is `true`.

crono_bool_t **`stop_on_timeout`**

Determine if the averager stops on timeout.

The timeout time is configured by *timeout_threshold*.

Default is `false`.

int **`timeout_threshold`**

Set the number of microseconds until timeout.

Must be 0 if no averaging application is installed on the Ndigo6G-12 board.

Default is 0.

struct **`ndigo6g12_tdc_channel`**

Contains TDC channel settings.

***Public Members***

crono_bool_t **`enable`**

Enable TDC channel.

Default is `false`.

crono_bool_t **`reserved3`**

Reserved for future extension.

crono_bool_t **`reserved2`**

Reserved for future extension.

crono_bool_t **`reserved1`**

Reserved for future extension.

*ndigo6g12_tdc_gating_block* **`gating_block`**

Configuration of the gating blocks.

*ndigo6g12_tdc_tiger_block* **`tiger_block`**

Configuration of the TiGer blocks.

struct **`ndigo6g12_tdc_gating_block`**

Contains settings of the gating blocks specifically for the TDCs.

The functionality is similiar to *ndigo6g12_gating_block*.

*Public Members*

crono_bool_t **enable**

Activates gating block.

crono_bool_t **negate**

Inverts output polarity.

Default is `false`.

crono_bool_t **retrigger**

Enable retriggering.

If enabled and a second trigger event is detected before the timer reaches `stop`, the timer is restarted. Otherwise signals at the input `sources` are ignored until `stop` is reached.

Defaults to `false`.

int **start**

The time from the first input signal seen in the idle state until the gating output is set.

In multiples of 5 ns. $0 \leq$ `start` $< 2^{16}$, while `start` $\leq$ `stop`.

Default is 0.

int **stop**

The number of samples from leaving the idle state until the gating output is reset.

In multiples of 5 ns. $0 \leq$ `stop` $< 2^{16}$, while `stop` $\geq$ `start`.

Default is 1000.

int **sources**

*Bit mask* with a bit set for all trigger sources that can trigger this channel.

Default *NDIGO6G12_TRIGGER_SOURCE_0* (*NDIGO6G12_TRIGGER_SOURCE_A0* for ADC channel A, *NDIGO6G12_TRIGGER_SOURCE_B0* for ADC channel B, etc).

struct **ndigo6g12_tdc_tiger_block**

Contains settings of TiGer block.

The configuration is similiar to *ndigo6g12_gating_block*.

*Public Members*

int **mode**

Enables the desired mode of operation for the TiGer.

Default is *NDIGO6G12_TIGER_OFF*.

Must be one of the following:

**NDIGO6G12_TIGER_OFF**

TiGer deactivated.

**NDIGO6G12_TIGER_OUTPUT**

Pulse height is approximately 2 V.

Connected hardware must not drive any signals to the connectors used as outputs, as doing so could damage both the Ndigo6G-12 and the external hardware. We recommend to only use short pulses to avoid undesirable baseline shift due to the AC coupling, but the device does not pose any restrictions on the duty cycle. This mode can be used as a clock output with a frequency of 75/N MHz (for integer N).

**NDIGO6G12_TIGER_BIDI**

Pulse height is approximately 1 V.

The LEMO connector may be used as input with OR function. Use short pulses to keep the probability of collision and the effect on the baseline low.

**NDIGO6G12_TIGER_BIPOLAR**

TiGer pulses are bipolar.

Not supported for inputs TRG and GATE.

In this mode, the connector creates bipolar pulses with 1 V amplitude. The connector can still be used as an input. Pulses have no effect on the baseline offset.

The TiGer should be configured with `start= stop + 1` for minimium-width bipolar pulses. The maximum bipolar pulse width is NDIGO6G12_TIGER_MAX_BIPOLAR_PULSE_LENGTH.

crono_bool_t **negate**

Set pulse polarity.

The TiGer creates a high pulse from `start` to `stop` unless negated.

Default is `true`.

crono_bool_t **retrigger**

Enable retriggering.

If enabled and a second trigger event is detected before the timer reaches `stop`, the timer is restarted. Otherwise signals at the input `sources` are ignored until `stop` is reached.

Defaults to `false`.

int **start**

The time from the first input signal seen in the idle state until the TiGer outputs a signal.

In multiples of 5 ns. $0 \le$ `start` $< 2^{16}$, while `start` $\le$ `stop`.

Default is 0.

int **stop**

The number of samples from leaving the idle state until the TiGer output is reset.

In multiples of 5 ns. $0 \le$ `stop` $< 2^{16}$, while `stop` $\ge$ `start`.

Note that the maximum length for bipolar pulses is given by NDIGO6G12_TIGER_MAX_BIPOLAR_PULSE_LENGTH.

Default is 1.

int **sources**

*Bit mask* with a bit set for all trigger sources that can trigger this channel.

Default *NDIGO6G12_TRIGGER_SOURCE_0* (*NDIGO6G12_TRIGGER_SOURCE_A0* for ADC channel A, *NDIGO6G12_TRIGGER_SOURCE_B0* for ADC channel B, etc).

## 3.5 Runtime control

int **ndigo6g12_start_capture**(*ndigo6g12_device* *device)

Start data acquisition.

**Parameters**

`device` – **[in]** Pointer to the device.

**Returns**

See *Function return values*.

int **ndigo6g12_stop_capture**(*ndigo6g12_device* *device)

Stop data acquisition.

**Parameters**

`device` – **[in]** Pointer to the device.

**Returns**

See *Function return values*.

int **ndigo6g12_manual_trigger**(*ndigo6g12_device* *device, int channel_mask)

Enables manual triggering of the ADC channels.

**Parameters**

- `device` – **[in]** Pointer to the device.
- `channel_mask` – **[in]** A bit mask that chooses which channels to trigger.

**Returns**

See *Function return values*.

int **ndigo6g12_single_shot**(*ndigo6g12_device* *device, int channel_mask)

Enables single-shot recording of the ADC channels.

Instead of continously triggering on input signals, only trigger and record a *ndigo6g12_trigger_block::multi_shot_count* number of events.

Note: Up to firmware revision 1.24120, this feature is bugged in 4-channel mode while *ndigo6g12_trigger_block::multi_shot_count* >1.

Requires that *ndigo6g12_trigger_block::enabled* is `false`.

**Parameters**

- `device` – **[in]** Pointer to the device.

- `channel_mask` – **[in]** A bit mask that chooses which channels to trigger.

**Returns**

See *Function return values*.

int **ndigo6g12_clear_pcie_errors**(*ndigo6g12_device* *device, int flags)

Clear PCIe errors.

Only useful for PCIe problem debugging flags.

**Parameters**

- `device` – **[in]** Pointer to the device.
- `flags` – **[in]** Specify which flags to clear.
  - *CRONO_PCIE_CORRECTABLE_FLAG*: clear all correctable errors
  - *CRONO_PCIE_UNCORRECTABLE_FLAG*: clear all uncorrectable errors

**Returns**

char array containing the plain text error message.

Relevant defines:

```
CRONO_PCIE_CORRECTABLE_FLAG
```

```
CRONO_PCIE_UNCORRECTABLE_FLAG
```

## 3.6 Readout

After an Ndigo6G-12 board is initialized and capturing, the captured events can be read from the board with `ndigo6g12_read()`. The read-out data is packaged in *packets* (see Chapter 4).

int **ndigo6g12_read**(*ndigo6g12_device* *device, *ndigo6g12_read_in* *in, *ndigo6g12_read_out* *out)

Reads packets from the board.

If *ndigo6g12_read_in::acknowledge_last_read* is `true`, automatically acknowledges the last read packets.

**Parameters**

- `device` – **[in]** Pointer to the device that should be read.
- `in` – **[in]** Pointer to the structure that configures the read call.
- `out` – **[out]** Pointer to a structure in which the read-out should be stored.

**Returns**

See *Function return values*.

const char ***ndigo6g12_get_last_error_message**(*ndigo6g12_device* *device)

Gets latest error message of **device**.

**Parameters**

`device` – **[in]** Pointer to the device.

**Returns**

char array containing the plain text error message.

const char ***ndigo6g12_device_state_to_str**(int state)

Convert **state** to plain text.

**Parameters**

**state** – **[in]** The device state as stored in *ndigo6g12_fast_info::state*.

**Returns**

char array containing the state as plain text.

struct **ndigo6g12_read_in**

The parameters of the read commands.

***Public Members***

crono_bool_t **acknowledge_last_read**

Automatically acknowledge packets from the previous call of *ndigo6g12_read*.

Only acknowledged packets will release the memory of the DMA buffer.

struct **ndigo6g12_read_out**

Struct for the read-out of the Ndigo6G-12 packets.

***Public Members***

volatile *crono_packet* ***first_packet**

Pointer to the first packet.

That is, the pointer that was captured by the call of *ndigo6g12_read*.

volatile *crono_packet* ***last_packet**

Pointer to the last packet.

int **error_code**

Error code.

Is one of the following:

`CRONO_READ_OK`

Reading packets from the device was successful.

`CRONO_READ_NO_DATA`

Trying to read packets does not yield data.

`CRONO_READ_INTERNAL_ERROR`

Some unhandled error occured. A device reinit is required.

`CRONO_READ_TIMEOUT`

Trying to read packets does not yield data in the given amount of time.

const char ***error_message**

Plain text error message.

# 4 Packet Format

Packets are retrieved by *ndigo6g12_read()*. They are of type *crono_packet*.

- Each hit on an ADC channel is stored in one packet. The format of the payload data (see *crono_packet::data*) is explained in Section 4.3.
- All TDC hits within the time given by *ndigo6g12_param_info::tdc_rollover_period* are stored in a single packet (stored in the payload data). The memory layout thereof is shown in Section 4.4.

> **Note**
>
> The minimum packet length depends on the application type. See Section 2.1 for details.

## 4.1 Output Structure *crono_packet*

struct **crono_packet**

uint8_t **channel**

Source channel of the data.

Values correspond to the following:

0x0 – 0x3: ADC A – D

≥ 0x4: TDC channels. Which specific TDC channel is encoded in *crono_packet::data*.

uint8_t **card**

ID of the card.

uint8_t **type**

Type of the packet.

Different packet types correspond to different encodings of *crono_packet::data*.

Is one of the following:

**CRONO_PACKET_TYPE_16_BIT_SIGNED**

Used for *ADC data*.

*crono_packet::data* must be cast to int16_t and *crono_packet::length* must be multiplied by 4.

**CRONO_PACKET_TYPE_TDC_DATA**

Used for *TDC data*.

*crono_packet::data* must be cast to uint32_t and *crono_packet::length* must be multiplied by 2.

**CRONO_PACKET_TYPE_AVRG_DATA**

Used for *averaged ADC data*.

*`crono_packet::data`* must be cast to `uint32_t` and *`crono_packet::length`* must be multiplied by 2.

uint8_t **`flags`**

Bit field of the following flags:

**`CRONO_PACKET_FLAG_SHORTENED`**

Packet was truncated because internal FIFO was full.

This means that less than the requested number of samples have been written.

**`CRONO_PACKET_FLAG_PACKETS_LOST`**

Lost triggers preceeded this packet due to insufficient DMA buffers.

The DMA controller has discarded packets due to the full host buffer.

**`CRONO_PACKET_FLAG_OVERFLOW`**

The packet contains ADC sample overflows.

**`CRONO_PACKET_FLAG_TRIGGER_MISSED`**

Lost triggers preceeded this packet due to insufficient buffers.

The trigger unit has discarded packets due to a full FIFO.

**`CRONO_PACKET_FLAG_DMA_FIFO_FULL`**

The internal DMA FIFO was full.

Triggers only got lost if a subsequent package has crono_packet::flags with a bit weight *CRONO_PACKET_FLAG_TRIGGER_MISSED* set.

**`CRONO_PACKET_FLAG_HOST_BUFFER_FULL`**

The host buffer was full.

Triggers only got lost if a subsequent package has crono_packet::flags with a bit weight *CRONO_PACKET_FLAG_TRIGGER_MISSED* set.

**`CRONO_PACKET_FLAG_TDC_NO_EDGE`**

The packet from a TDC does not contain valid data.

Hence, the timestamp is not corrected. No valid edge was found for the TDC.

For TDC data, can also be one of the following:

**`NDIGO6G12_TDC_PACKET_FLAG_RESERVED`**

**`NDIGO6G12_TDC_PACKET_FLAG_CONTAINS_DATA`**

Packet contains at least one TDC event.

**`NDIGO6G12_TDC_PACKET_FLAG_LOST`**

At least one packet was lost due to full FIFO.

`NDIGO6G12_TDC_PACKET_FLAG_SHORTENED`

The trigger unit has shortend the current packet due to full FIFO.

`NDIGO6G12_TDC_PACKET_FLAG_DMA_FIFO_FULL`

The DMA FIFO was full.

Trigger only got lost if a subsequent package has crono_packet::flags with a bit weight *NDIGO6G12_TDC_PACKET_FLAG_LOST* set.

`NDIGO6G12_TDC_PACKET_FLAG_HOST_BUFFER_FULL`

The host buffer was full.

Trigger only got lost if a subsequent package has crono_packet::flags with a bit weight *NDIGO6G12_TDC_PACKET_FLAG_LOST* set.

uint32_t **length**

Length of *crono_packet::data* in multiples of 64 bits. The actual length of *crono_packet::data* depends on *crono_packet::type*.

uint64_t **timestamp**

Timestamp of the packet.

For the Ndigo6G-12, this corresponds to the beginning of the packet data.

uint64_t **data**[1]

Payload data of the packet.

The length of `data` corresponds to *crono_packet::length*.

The data type must be cast according to *crono_packet::type*, and the data encoding also depends on *crono_packet::type*.

See Section 4.3 for the data encoding of ADC data.

See Section 4.4 for the data encoding of TDC data.

See Section 4.5 for the data encoding of averaged ADC data.

## 4.2 Utility macros

The following macros can be used to navigate through the packets obtained by *ndigo6g12_read()*.

**crono_packet_data_length**(current)

Returns the length of crono_packet::data in multiples of 8 bytes.

**crono_packet_bytes**(current)

Returns the length of crono_packet::data including its header in bytes.

**crono_next_packet**(current)

Returns a crono_packet pointer pointing to the next packet in the host buffer.

Must be checked before use to not point beyond the last packet of the readout data, e.g., `crono_next_packet(current_packet) <= readout_data.last_packet`.

## 4.3 Data encoding for ADC hits

*data*, that is, the packet-data payload, depends on *ndigo6g12_configuration::output_mode*. The length of the *data* array is encoded in *length*. Be aware that *length* is in multiples of 64 bit, while the size of the fields of *data* depends on *type*.

Thus, reading packet data requires the following steps:

- Depending on *crono_packet::type*, multiply *length* appropriately. E.g., if *type* is *CRONO_PACKET_TYPE_16_BIT_SIGNED*, *length* has to be multiplied by 4 (since 4 × 16 bit = 64 bit).
- Cast *data* according to *type*. E.g., if *type* is *CRONO_PACKET_TYPE_16_BIT_SIGNED*, cast *data* to `int16_t`.

### 4.3.1 NDIGO6G12_OUTPUT_MODE_SIGNED16

Raw data of the ADC is mapped to the range of a signed16 integer (−32768 to 32767). Packet data must be cast to `int16_t`.

### 4.3.2 NDIGO6G12_OUTPUT_MODE_RAW

Packet data is returned in the native range of the ADC (0 to 4095). It must be cast to `int16_t`.

Data layout:

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | ... | 0 |
|---|---|---|---|---|---|---|---|---|
| Data | 0 | 0 | control bits | | sample data | | | |

### 4.3.3 NDIGO6G12_OUTPUT_MODE_RAW_NO_CB

Packet data is returned in the native range of the ADC (0 to 4095). It must be cast to `int16_t`. Unlike *NDIGO6G12_OUTPUT_MODE_RAW*, it does not contain control bits.

> **Attention**
>
> *NDIGO6G12_OUTPUT_MODE_RAW* and *NDIGO6G12_OUTPUT_MODE_RAW_NO_CB* are useful for debugging purposes. They are not supported for user applications. Use *NDIGO6G12_OUTPUT_MODE_SIGNED16* instead.

### 4.3.4 NDIGO6G12_OUTPUT_MODE_SIGNED32

Only used if *ndigo6g12_init_parameters::application_type* is *NDIGO6G12_APP_TYPE_AVRG*. See Section 4.5 for more information.

## 4.4 Data encoding for TDC hits

The following bit table shows the encoding of the payload data (*crono_packet::data*) of all recorded TDC hits within the time-frame given by *ndigo6g12_param_info::tdc_rollover_period*.

| Bit | 31 | 30 | ... | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Data | Timestamp | | | | | TDC hit flags | | | | Channel number | | | |

Details:

- The timestamp is relative to *crono_packet::timestamp* and is given in units of *ndigo6g12_param_info::tdc_period*.
- The channel numbers are:
  0x0: TDC channel 1
  0x1: TDC channel 2
  0x2: TDC channel 3
  0x3: TDC channel 4
  0x4: TRG
  0x5: GATE
  0xD: Dummy data
  0xF: Rollover marker

The TDC hit flags are one of the following:

**NDIGO6G12_TDC_HIT_FLAG_LOST**

At least one preceding event was lost due to full FIFO.

**NDIGO6G12_TDC_HIT_FLAG_ROLLOVER_LOST**

Rollover has been lost due to full FIFO.

Results in a fatal error.

**NDIGO6G12_TDC_HIT_FLAG_VALID**

Timestamp is a valid TDC event.

**NDIGO6G12_TDC_HIT_FLAG_GROUP_TIME_ROLLOVER**

Timestamp is a rollover marker.

Add *ndigo6g12_param_info::tdc_rollover_period* to all subsequent timestamps in the packet.

**NDIGO6G12_TDC_HIT_ERROR_MASK**

TDC hit flag mask for error reporting.

**NDIGO6G12_TDC_HIT_TYPE_MASK**

TDC hit flags mask for timestamp type.

**NDIGO6G12_TDC_PADDING_DATA_CHANNEL**

TDC hit channel number for padding-data.

Padding-data can be ignored. Does not contain any usefull information. Padding-data has *NDIGO6G12_TDC_HIT_FLAG_GROUP_TIME_ROLLOVER* and *NDIGO6G12_TDC_HIT_FLAG_VALID* always cleared.

**NDIGO6G12_TDC_ROLLOVER_CHANNEL**

TDC hit channel number for rollover marker.

Rollover marker has *NDIGO6G12_TDC_HIT_FLAG_GROUP_TIME_ROLLOVER* always set.

## 4.5 Data encoding for averaged ADC hits

When using *NDIGO6G12_APP_TYPE_AVRG*, the first two 64-bit words of *crono_packet::data* are the extended header containing more information, as shown in the following bit table.

| Bits | 127 – 38 | 27 – 32 | 31 – 12 | 11 – 0 |
|---|---|---|---|---|
| Data | reserved | flags | reserved | iterations performed |

Flags:

- 0x01: stopped iterations prematurely
- 0x02: overflow detected
- 0x04: stopped by timeout
- 0x08: stopped by software
- 0x10: stopped by overflow

The following data words contain the raw ADC data mapped to the range of a signed32 integer ($-2^{31}$ to $2^{31}-1$). Thus, *crono_packet::data* must be cast to int32_t and *crono_packet::length* must be multiplied by 2 **taking into account the extended header**. That is,

```
uint32_t extended_header_length = 2;
uint32_t sample_count = ((pkt->length - extended_header_length) * 2);
int32_t* adc_data = (int32_t*)(pkt->data + extended_header_length);
for (uint32_t i = 0; i < sample_count; i++) {
    /* work with adc_data[i] */
}
```

# 5 C++-Example

The following source code is an example of an Ndigo6G-12 application written in C++. The source code is also available on our GitHub.

| Source file | Description |
|---|---|
| *ndigo6g12_example.cpp* | Main source-code file of the example application. |
| *ndigo6g12_app.h* | Header file for classes for different Ndigo6G-12 `application types` and TDC setup. |
| *ndigo6g12_adc_single.cpp* | Implementation of application type `NDIGO6G12_APP_TYPE_1CH`. |
| *ndigo6g12_adc_dual.cpp* | Implementation of application type `NDIGO6G12_APP_TYPE_2CH`. |
| *ndigo6g12_adc_quad.cpp* | Implementation of application type `NDIGO6G12_APP_TYPE_4CH`. |
| *ndigo6g12_adc_averager.cpp* | Implementation of application type `NDIGO6G12_APP_TYPE_AVRG`. |
| *ndigo6g12_tdc.cpp* | Implementation of the TDC-class. |
| *delay.h* | Implementation for measuring delays. |

## 5.1 ndigo6g12_example.cpp

```cpp
// Example application for the Ndigo6G-12
//
#include "ndigo6g12_app.h"
#include "ndigo6g12_interface.h"
#include <map>
#include <stdio.h>
#include <stdlib.h>

std::map<int, std::string> appTypeMap = {{1, "One ADC channels @6.4 Gsps"},
                                         {2, "Two ADC channels @3.2 Gsps"},
                                         {4, "Four ADC channels @1.6 Gsp"},
                                         {5, "Averaging mode @6.4 Gsps"}};

std::map<int, std::string> requirementsMap = {
    {0,
     "Starts the test of the currently configured app type"},
    {1,
     "Measure time distance between passing of "
        "threshold, calculates the frequency, requires NIM signal on channel A"},
    {2, "Dual-channel application that measures delay between start "
        "pulse on channel A and stop pulse on channel D (NIM)"},
```

(continues on next page)

(continued from previous page)

```
    {4, "Quad-channel application that measures delay between start "
        "pulse on channel A and stop pulses on channels B-D (NIM)"},
    {5, "Measure time distance between averaged start on TRG (NIM) and stop on "
        "channel A (falling) by summing data of 16 runs "
        "to increase precision of measurement for signal with low amplitude"}};

Ndigo6GApp *adcApp;
ndigo6g12_param_info paramInfo;

// initialize Ndigo6G-12 device
ndigo6g12_device initialize_ndigo6g12(int bufferSize, int boardId,
                                      int cardIndex, int appType, int tdcChannels) {
    // prepare initialization
    ndigo6g12_init_parameters params;
    // fill initialization data structure with default values
    // so that the data is valid and only parameters
    // of interest have to be set explicitly
    ndigo6g12_get_default_init_parameters(&params);
    params.application_type = appType;

    params.buffer_size[0] = bufferSize; // size of the packet buffer
    params.board_id = boardId; // value copied to "card" field of every packet,
                               // allowed range 0..255
    params.card_index = cardIndex; // which of the Ndigo6G-12 board found in
                                   // the system to be used
    // this specifies the directories or the specific .cronorom if dynamic
    // switching of appType is required. If not specified, the example will
    // return an error if the appType does not match the current appType in the
    // firmware
    params.firmware_locations = ".";

    // initialize card
    int errorCode;
    const char *errorMessage;
    ndigo6g12_device device;
    errorCode = ndigo6g12_init(&device, &params, &errorMessage);

    if (errorCode != CRONO_OK) {
        printf("Could not init Ndigo6G-12: %s\n", errorMessage);
        printf("Please change path to the .cronorom in ndigo6g12_example.cpp\n");
        exit(1);
    }

    // check if firmware now supports the chosen application type
    ndigo6g12_static_info si;
    ndigo6g12_get_static_info(&device, &si);
    if (si.application_type != appType) {
        printf("The switch to appType did not work, please make sure that "
               "the firmware file is provided");
```

(continues on next page)

(continued from previous page)

```
        ndigo6g12_close(&device);
        exit(1);
    }
    if (appType == 0) {
        appType = si.application_type;
    }
    switch (appType) {
    case 1:
        adcApp = new Ndigo6GAppSingle(tdcChannels);
        break;
    case 2:
        adcApp = new Ndigo6GAppDual(tdcChannels);
        break;
    case 4:
        adcApp = new Ndigo6GAppQuad(tdcChannels);
        break;
    case 5:
        adcApp = new Ndigo6GAppAverager(tdcChannels);
        break;
    default:
        printf("Not supported appType %d'\n", appType);
        ndigo6g12_close(&device);
        exit(1);
    }
    printf("Running in %s\n%s\n", appTypeMap[appType].c_str(),
           requirementsMap[appType].c_str());
    return device;
}

int configure_ndigo6g12(ndigo6g12_device *device, int adcThreshold) {
    // prepare configuration
    ndigo6g12_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
    if (CRONO_OK != ndigo6g12_get_default_configuration(device, &config)) {
        printf("Could not get default configuration: %s\n",
               ndigo6g12_get_last_error_message(device));
        ndigo6g12_close(device);
        return 1;
    }

    //**********************************************************************
    // configuration for the TDC channels
    adcApp->ConfigureTDC(&config);

    //**********************************************************************
    // configuration for the ADC channels
    adcApp->ConfigureADC(&config, adcThreshold);
```

(continues on next page)

(continued from previous page)

```

    // write configuration to board
    int error_code = ndigo6g12_configure(device, &config);
    if (error_code != CRONO_OK) {
        printf("Could not configure Ndigo6G-12: %s\n",
               ndigo6g12_get_last_error_message(device));
        return 1;
    }
    ndigo6g12_get_param_info(device, &paramInfo);
    adcApp->SetParamInfo(&paramInfo);
    return 0;
}

// print some basic information about the Ndigo6G-12 device
void print_device_information(ndigo6g12_device *device) {
    ndigo6g12_static_info si;
    ndigo6g12_get_static_info(device, &si);
    printf("Firmware revision %d.%d - Type %d\n", si.fw_revision,
           si.svn_revision, si.application_type);
    printf("Firmware Bitstream Timestamp : %s\n", si.bitstream_date);
    printf("Calibration date             : %s\n", si.calibration_date);
    printf("Board serial                 : %d.%d\n", si.board_serial >> 24,
           si.board_serial & 0xffffff);
    printf("Board revision               : %d\n", si.board_revision);
    printf("Board configuration          : %d\n", si.board_configuration);
    printf("Driver Revision              : %d.%d.%d\n",
           ((si.driver_revision >> 16) & 255),
           ((si.driver_revision >> 8) & 255), (si.driver_revision & 255));
    printf("Driver Build Revision        : %d\n", si.driver_build_revision);

    ndigo6g12_fast_info fi;
    ndigo6g12_get_fast_info(device, &fi);
    printf("TDC temperature              : %.2f C\n", fi.tdc1_temp);
    printf("ADC temperature              : %.2f C\n", fi.ev12_temp);
    printf("FPGA temperature             : %.2f C\n", fi.fpga_temperature);
    printf("PCIe link speed              : Gen. %d\n", fi.pcie_link_speed);
    printf("PCIe link width              : %d lanes\n", fi.pcie_link_width);
    printf("PCIe payload                 : %d bytes\n", fi.pcie_max_payload);

    ndigo6g12_param_info pi;
    ndigo6g12_get_param_info(device, &pi);
    printf("Sample rate                  : %.0f Msps\n",
           pi.sample_rate / 1000000.0);
    printf("Resolution                   : %d Bit\n", pi.resolution);
    printf("Sample period                : %.2f ps\n", pi.sample_period);
    printf("TDC bin size                 : %.2f ps\n", pi.tdc_period);
    printf("Packet Timestamp period      : %.2f ps\n", pi.packet_ts_period);
    printf("ADC Sample delay             : %.2f ps\n", pi.adc_sample_delay);
}

```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: ndigo6g12_example <appType> [<tdcMask>]\n");
        for (auto atPair : appTypeMap) {
            int at = atPair.first;
            printf("AppType %d: %s\n  %s\n", at, appTypeMap[at].c_str(),
                   requirementsMap[at].c_str());
        }
        printf("tdcMask: Bit flag for TDC channels E-H\n");
        exit(1);
    }

    int appType = atoi(argv[1]);
    int tdcChannelMask = 0;
    if (argc > 2) {
        tdcChannelMask = atoi(argv[2]);
    }
    // use 128 MiByte to buffer incoming data
    // largest ADC data packet has about 500 KiByte
    const int64_t BUFFER_SIZE = 128 * 1024 * 1024;

    // use the first Ndigo6G-12 device found in the system
    const int CARD_INDEX = 0;

    // set board ID in all packets to 0
    // can be used to distinguish packets of multiple devices
    const int BOARD_ID = 0;

    printf("cronologic ndigo6g12_example using driver: %s\n",
           ndigo6g12_get_driver_revision_str());

    // create and initialize the device
    // may fail if the device is already in use by another process
    // or the device driver is not installed
    ndigo6g12_device device =
        initialize_ndigo6g12(BUFFER_SIZE, BOARD_ID, CARD_INDEX, appType,
                             tdcChannelMask);

    print_device_information(&device);

    // set the configuration required for capturing data
    // the base line is shifted by +350mV, as the target is to trigger at the
    // middle of the NIM pulse edge
    int adcThreshold = 0;
    int status = configure_ndigo6g12(&device, adcThreshold);
    if (status != 0) {
        exit(1);
    }

    // configure readout behaviour
```

(continues on next page)

(continued from previous page)

```
    // automatically acknowledge all data as processed
    // on the next call to ndigo6g12_read()
    // old packet pointers are invalid after calling ndigo6g12_read()
    ndigo6g12_read_in readConfig;
    readConfig.acknowledge_last_read = 1;

    // structure with packet pointers for read data
    ndigo6g12_read_out readData;

    // start data capture
    status = ndigo6g12_start_capture(&device);
    if (status != CRONO_OK) {
        printf("Could not start capturing: %s",
               ndigo6g12_get_last_error_message(&device));
        ndigo6g12_close(&device);
        exit(1);
    }

    // get current sample rate to calculate event timestamps
    ndigo6g12_param_info paramInfo;
    ndigo6g12_get_param_info(&device, &paramInfo);

    // ADC data is provided in packets, one packet per ADC channel and trigger
    // TDC data is provided in a single packet for all TDC inputs in a certain
    // timespan
    printf("\nReading packets:\n");

    const int MAX_PACKET_COUNT = 70;
    int packetCount = 0;
    bool noDataLastTime = false;
    while ((packetCount < MAX_PACKET_COUNT)) {
        // get pointers to acquired packets
        status = ndigo6g12_read(&device, &readConfig, &readData);
        if (status != CRONO_OK) {
            if (!noDataLastTime) {
                printf(" - No data! -\n");
            }
            noDataLastTime = true;

        } else {
            noDataLastTime = false;
            // iterate over all packets received by the last read
            volatile crono_packet *p = readData.first_packet;
            while (p <= readData.last_packet) {

                if (p->channel < 4) {
                    // packets with channel number < 4 are ADC data
                    double packet_ts =
                        adcApp->ProcessADCPacket(const_cast<crono_packet *>(p));
                } else {
```

(continues on next page)

(continued from previous page)

```
                    // packets with channel number >= 4 are TDC data
                    adcApp->ProcessTDCPacket(const_cast<crono_packet *>(p));
                }

                // go to next packet
                p = crono_next_packet(p);

                packetCount++;
            } // end: iterate over received packets
        }     // end: Got any packets?
    }         // end: while

    // shut down packet generation and DMA transfers
    ndigo6g12_stop_capture(&device);

    // deactivate Ndigo6G-12
    ndigo6g12_close(&device);

    return 0;
}
```

## 5.2 ndigo6g12_app.h

```
#pragma once
#include "delay.h"
#include "ndigo6g12_interface.h"
#include <map>
#include <string>
#include <vector>

// Base class for Ndigo6G applications
// contains common code for packet processing
class Ndigo6GApp {
  protected:
    const int PRECURSOR = 1;
    // contains the timing parameters of the current mode like sample period
    ndigo6g12_param_info *pi;
    int adcThreshold;
    int tdcChannelMask;
    // convenience method for adding the TDC channels to the channel map
    void AddTDCChannels(std::map<int, std::string> &channelMap) {
        for (int i = 0; i < 4; i++) {
            if (tdcChannelMask & (1 << i)) {
                channelMap[4 + i] = (char)'E' + (char)i;
            }
        }
    }

  public:
```

(continues on next page)

(continued from previous page)

```
    Ndigo6GApp(int tdcChannelMask) { this->tdcChannelMask = tdcChannelMask; }

    // configure the Ndigo6G with the appropiate mode and triggers
    virtual void ConfigureADC(ndigo6g12_configuration *config,
                              int adcThreshold) = 0;

    virtual void ConfigureTDC(ndigo6g12_configuration *config);

    // react to an ADC incoming packet
    virtual double ProcessADCPacket(crono_packet *pkt) = 0;

    // set the parameters after configuration was successful
    virtual void SetParamInfo(ndigo6g12_param_info *pi) { this->pi = pi; }

    // called by the main loop on a TDC packet arrival
    virtual void ProcessTDCPacket(crono_packet *pkt);

    // react to an incoming TDC packet, called by default implementation of
    // ProcessTDCPacket
    virtual void ProcessTDCTimestamp(int tdcChannel, double timestamp) {
        printf("TDC event on channel %d timestamp: %.3f ns\n", tdcChannel,
               timestamp / 1000.0);
    };

    // helper method to find the timestamp of the current packet
    double ComputePacketTimestamp(volatile crono_packet *pkt) {
        // calculate packet timestamp in picoseconds
        // the precursor time is constant in the modes, but the amount of
        // samples is different (32/16/8 for 1/2/4)
        double packet_ts =
            pkt->timestamp * pi->packet_ts_period - PRECURSOR * 5e3 ;
        return packet_ts;
    }

    // Computes the falling edge in the given data, returns the absolute ps
    // value, and -1 if threshold was not passed in the packet.
    double ComputeFallingEdge(crono_packet *pkt) {
        // packet length is number of 64-bit words of data
        double packetTs = ComputePacketTimestamp(pkt);
        // 4 ADC samples are stored in each 64-bit chunk of packet data
        uint32_t sampleCount = (pkt->length * 4);

        // ADC data is a signed 16-bit integer
        int16_t *adc_data = (int16_t *)(pkt->data);

        // find first falling edge in ADC data
        for (uint32_t i = 0; i < (sampleCount - 1); i++) {
            if (adc_data[i] >= adcThreshold && adc_data[i + 1] < adcThreshold) {
                // calculate threshold crossing relative to start of packet
                double feOffset = i;
```

(continues on next page)

(continued from previous page)

```
        // linear interpolation of trigger threshold crossing
        feOffset += (double)(adc_data[i] - adcThreshold) /
              (adc_data[i] - adc_data[i + 1]);
        // convert to picoseconds
        feOffset *= pi->sample_period;

        // calculate timestamp of threshold crossing in picoseconds
        double fallingEdgeTs = packetTs + feOffset;
        // adjust for ADC pipeline delay
        fallingEdgeTs -= pi->adc_sample_delay;

        return fallingEdgeTs;
      }
    }
    return -1;
  }


};
// maximum distance of two pulses, so that they are considered to be a cable delay
static const double MAX_DELAY_PS = 500000.;

class Ndigo6GAppSingle : public Ndigo6GApp {
 private:
  // last falling edge to compute the difference to
  double lastFallingEdgeTs = 0;

 public:
  Ndigo6GAppSingle(int tdcChannelMask) : Ndigo6GApp(tdcChannelMask) {
  }
  virtual void ConfigureADC(ndigo6g12_configuration *config,
               int adc_threshold);
  virtual double ProcessADCPacket(crono_packet *pkt);
  virtual void ProcessTDCTimestamp(int tdcChannel, double timestamp) {}
};



// Implementation of the different sample applications
class Ndigo6GAppDual : public Ndigo6GApp {
 private:
  DelayMeasurement delayMeasure;

 public:
  Ndigo6GAppDual(int tdcChannelMask) : Ndigo6GApp(tdcChannelMask) {
    std::map<int, std::string> channelMap = {{0, "A"}, {3, "D"}};
    AddTDCChannels(channelMap);
    delayMeasure.Init(0, MAX_DELAY_PS, channelMap);
  }
```

(continues on next page)

(continued from previous page)

```
    virtual void ConfigureADC(ndigo6g12_configuration *config,
                              int adc_threshold);

    virtual double ProcessADCPacket(crono_packet *pkt);

    virtual void ProcessTDCTimestamp(int tdcChannel, double timestamp);
    virtual void SetParamInfo(ndigo6g12_param_info *pi) {
        Ndigo6GApp::SetParamInfo(pi);
        // we have to wait for 3 TDC periods to make sure that the TDC data has
        // arrived
        delayMeasure.SetMaxWaitTime(pi->tdc_rollover_period * 3.5 *
                                    pi->tdc_period);
    }
};

class Ndigo6GAppQuad : public Ndigo6GApp {
  private:
    DelayMeasurement delayMeasure;

  public:
    Ndigo6GAppQuad(int tdcChannelMask) : Ndigo6GApp(tdcChannelMask) {

        std::map<int, std::string> channelMap = {
            {0, "A"}, {1, "B"}, {2, "C"}, {3, "D"}};
        AddTDCChannels(channelMap);
        delayMeasure.Init(0, MAX_DELAY_PS, channelMap);
    }

    virtual void ConfigureADC(ndigo6g12_configuration *config,
                              int adc_threshold);

    virtual  void SetParamInfo(ndigo6g12_param_info *pi) {
        Ndigo6GApp::SetParamInfo( pi);
        // we have to wait for 3 TDC periods to make sure that the TDC data has
        // arrived
        delayMeasure.SetMaxWaitTime(pi->tdc_rollover_period * 3.5 *
                                    pi->packet_ts_period);
    }

    virtual double ProcessADCPacket(crono_packet *pkt);

    virtual void ProcessTDCTimestamp(int tdcChannel, double timestamp);
};

class Ndigo6GAppAverager : public Ndigo6GApp {
  private:
    // last falling edge to compute the difference to
    double lastFallingEdgeTs = 0;

  public:
```

(continues on next page)

(continued from previous page)

```
    Ndigo6GAppAverager(int tdcChannelMask) : Ndigo6GApp(tdcChannelMask) {}
    virtual void ConfigureADC(ndigo6g12_configuration *config,
                              int adc_threshold);
    virtual double ProcessADCPacket(crono_packet *pkt);
};
```

## 5.3 ndigo6g12_adc_single.cpp

```
#include "ndigo6g12_app.h"
#include <stdio.h>

// a simple application that measures the distance of two packets and computes
// the frequency of the signal
 double Ndigo6GAppSingle::ProcessADCPacket(crono_packet *pkt) {

    double fallingEdgeTs = ComputeFallingEdge(pkt);

    if (fallingEdgeTs > 0) {
        if (lastFallingEdgeTs > 0) {
            double packetRate = (1.0 / (fallingEdgeTs - lastFallingEdgeTs));
            double packetRateKHz = packetRate * 1e9;
            printf("ADC packet rate: %.3f kHz\n", packetRateKHz);
        }
        lastFallingEdgeTs = fallingEdgeTs;
    }
    return fallingEdgeTs;
}

void Ndigo6GAppSingle::ConfigureADC(ndigo6g12_configuration *config,
                                    int adcThreshold) {
    this->adcThreshold = adcThreshold;
    // single channel mode with 6.4 Gsps
    config->adc_mode = NDIGO6G12_ADC_MODE_A;

    // ADC sample value range -32768 .. 32767
    config->output_mode = NDIGO6G12_OUTPUT_MODE_SIGNED16;

    // enable ADC channel A and trigger on the falling edge of ADC data
    // shift baseline of analog inputs to +350 mV
    config->analog_offsets[0] = NDIGO6G12_DC_OFFSET_N_NIM * -1;

    // trigger on falling edge of ADC data
    config->trigger[NDIGO6G12_TRIGGER_A0].edge = true;
    config->trigger[NDIGO6G12_TRIGGER_A0].rising = false;
    config->trigger[NDIGO6G12_TRIGGER_A0].threshold = adcThreshold;

    // enable channel A
    config->trigger_block[0].enabled = true;
    // multiples of 32 ADC samples (5 ns recording time)
```

(continues on next page)

(continued from previous page)

```
    config->trigger_block[0].length = 1;
    // multiples of 32 ADC samples, gets added to packet length

    config->trigger_block[0].precursor = PRECURSOR;

    // select ADC data as trigger source of the channel
    config->trigger_block[0].sources = NDIGO6G12_TRIGGER_SOURCE_A0;

}
```

## 5.4 ndigo6g12_adc_dual.cpp

```
#include "ndigo6g12_app.h"
#include <stdio.h>
#include <cmath>


// an application that measures the delay between a start signal (A) and a
// stop signal (D)
double Ndigo6GAppDual::ProcessADCPacket(crono_packet *pkt) {

    double falling_edge_ts = ComputeFallingEdge(pkt);

    // gather data
    if (falling_edge_ts > 0) {
        delayMeasure.InsertTimestamp(pkt->channel, falling_edge_ts);
    }
    Delays *delays = delayMeasure.MeasureDelays();

    delayMeasure.PrintDelays(delays);

    return falling_edge_ts;
}
void Ndigo6GAppDual::ProcessTDCTimestamp(int tdcChannel, double timestamp) {
    //TDC channels are mapped as 4-7
    delayMeasure.InsertTimestamp(4 + tdcChannel, timestamp);

    Delays *delays = delayMeasure.MeasureDelays();

    delayMeasure.PrintDelays(delays);
}


void Ndigo6GAppDual::ConfigureADC(ndigo6g12_configuration *config,
                                  int adcThreshold) {
    this->adcThreshold = adcThreshold;
    // dual channel mode with 3.2 Gsps
    config->adc_mode = NDIGO6G12_ADC_MODE_AD;

```

(continues on next page)

(continued from previous page)

```
    // ADC sample value range -32768 .. 32767
    config->output_mode = NDIGO6G12_OUTPUT_MODE_SIGNED16;

    // enable ADC channel A and trigger on the falling edge of ADC data
    // shift baseline of analog inputs to +350 mV
    // do the same for channel D
    config->analog_offsets[0] = NDIGO6G12_DC_OFFSET_N_NIM * -1;
    config->analog_offsets[3] = NDIGO6G12_DC_OFFSET_N_NIM * -1;

    // trigger on falling edge of ADC data
    config->trigger[NDIGO6G12_TRIGGER_A0].edge = true;
    config->trigger[NDIGO6G12_TRIGGER_A0].rising = false;
    config->trigger[NDIGO6G12_TRIGGER_A0].threshold = adcThreshold;
    config->trigger[NDIGO6G12_TRIGGER_D0].edge = true;
    config->trigger[NDIGO6G12_TRIGGER_D0].rising = false;
    config->trigger[NDIGO6G12_TRIGGER_D0].threshold = adcThreshold;

    // enable channel A
    config->trigger_block[0].enabled = true;

    // in multiples of 16 ADC samples (5 ns recording time)
    config->trigger_block[0].length = 1;

    // in multiples of 16 ADC samples, gets added to packet length
    config->trigger_block[0].precursor = PRECURSOR;

    // select ADC data as trigger source of the channel
    config->trigger_block[0].sources = NDIGO6G12_TRIGGER_SOURCE_A0;

    // enable channel D
    config->trigger_block[3].enabled = true;

    // in multiples of 16 ADC samples (5 ns recording time)
    config->trigger_block[3].length = 1;

    // in multiples of 16 ADC samples, gets added to packet length
    config->trigger_block[3].precursor = PRECURSOR;

    // select ADC data as trigger source of the channel
    config->trigger_block[3].sources = NDIGO6G12_TRIGGER_SOURCE_D0;
}
```

## 5.5 ndigo6g12_adc_quad.cpp

```
#include "ndigo6g12_app.h"
#include <stdio.h>
#include <cmath>
#include <array>

```

(continues on next page)

(continued from previous page)

```
// an application that measures the delay between a start signal (A)
// stop signals on channels B-D
double Ndigo6GAppQuad::ProcessADCPacket(crono_packet *pkt) {

    double falling_edge_ts = ComputeFallingEdge(pkt);

    // gather data
    if (falling_edge_ts > 0) {
        delayMeasure.InsertTimestamp(pkt->channel, falling_edge_ts);
    }

    Delays *delays = delayMeasure.MeasureDelays();

    delayMeasure.PrintDelays(delays);

    return falling_edge_ts;
}

void Ndigo6GAppQuad::ProcessTDCTimestamp(int tdcChannel, double timestamp) {
    // insert TDC as channel 4-7
    delayMeasure.InsertTimestamp(4 + tdcChannel, timestamp);

    Delays *delays = delayMeasure.MeasureDelays();

    delayMeasure.PrintDelays(delays);
}

void Ndigo6GAppQuad::ConfigureADC(ndigo6g12_configuration *config,
                                  int adcThreshold) {
    this->adcThreshold = adcThreshold;
    // quad channel mode with 1.6 Gsps
    config->adc_mode = NDIGO6G12_ADC_MODE_ABCD;

    // ADC sample value range -32768 .. 32767
    config->output_mode = NDIGO6G12_OUTPUT_MODE_SIGNED16;
    // trigger on falling edge of ADC data
    for (int index : {NDIGO6G12_TRIGGER_A0, NDIGO6G12_TRIGGER_B0,
                      NDIGO6G12_TRIGGER_C0, NDIGO6G12_TRIGGER_D0}) {
        config->trigger[index].edge = true;
        config->trigger[index].rising = false;
        config->trigger[index].threshold = adcThreshold;
    }

    // the sources of each channel (they should trigger on the input data
    // of the channel)
    std::array<int, 4> sources = {
        NDIGO6G12_TRIGGER_SOURCE_A0, NDIGO6G12_TRIGGER_SOURCE_B0,
        NDIGO6G12_TRIGGER_SOURCE_C0, NDIGO6G12_TRIGGER_SOURCE_D0};

```

(continues on next page)

(continued from previous page)

```
    // enable ADC channels A-D and trigger on the falling edge of ADC data
    // shift baseline of analog inputs to +350 mV
    for (int c = 0; c < 4; c++) {
        config->analog_offsets[c] = NDIGO6G12_DC_OFFSET_N_NIM * -1;

        // enable channel
        config->trigger_block[c].enabled = true;

        // in multiples of 8 ADC samples (5 ns recording time) after trigger
        config->trigger_block[c].length = 1;

        // in multiples of 8 ADC samples, gets added to packet length
        config->trigger_block[c].precursor = PRECURSOR;

        // select ADC data as trigger source of the channel
        config->trigger_block[c].sources = sources[c];
    }
}
```

## 5.6 ndigo6g12_adc_averager.cpp

```
#include <stdio.h>
#include "ndigo6g12_app.h"

const int AVERAGING_COUNT = 16;

double Ndigo6GAppAverager::ProcessADCPacket(crono_packet* pkt) {

        // calculate packet timestamp in picoseconds
        // not adjusted for ADC-data precursor
        double packet_ts = pkt->timestamp * pi->packet_ts_period;

        printf("\nPacket timestamp: %.3f ns\n", (packet_ts / 1000.0));

        // packet length is number of 64-bit words of data
        // the first two 64-bit packet data words are additional header
        // information
        uint32_t data_offset = 2;
        // only the first currently carries valid information
        uint64_t averaging_header0 = *(pkt->data);

        // if bit is set, less than the requested number of iterations have been
        // performed before writing the packet due to possible data overflow on
        // the next iteration
        bool stopped_due_to_overflow = (averaging_header0 >> 32) & 0x1;

        // if bit is set, the averaged data contains saturated or overflowed
        // samples does NOT indicate that the input signal has not exceeded the
        // ADC range
```

(continues on next page)

(continued from previous page)

```
        bool averaging_overflow = (averaging_header0 >> 32) & 0x2;

        // number of iterations; may be less than requested
        int iterations_performed = (averaging_header0 & 0xffffff);

        // 2 averaged ADC samples are stored in each 64-bit chunk of packet data
        uint32_t sample_count = ((pkt->length - data_offset) * 2);

        // ADC data is a signed 32-bit integer
        int32_t* adc_data = (int32_t*)(pkt->data + data_offset);

        // find first falling edge in averaging data
        for (uint32_t i = 0; i < sample_count - 1; i++) {
                if (adc_data[i] >= 0 && adc_data[i + 1] < 0) {
                        // calculate threshold crossing relative to start of packet
                        double fe_offset = i;
                        // linear interpolation of trigger threshold crossing
                        fe_offset +=
                                (double)(adc_data[i] - 0) / (adc_data[i] - adc_data[i + 1]);
                        // calculate timestamp of threshold crossing in picoseconds
                        fe_offset *= pi->sample_period;

                        printf("Falling edge event - offset to packet start: %.3f ns\n",
                                (fe_offset / 1000.0));
                        break;
                }
        }
        return packet_ts;
}

void Ndigo6GAppAverager::ConfigureADC(ndigo6g12_configuration* config,
                                                                              int adcThreshold) {
        // adcThreshold not used here, 0 is used as threshold for the data
        config->adc_mode = NDIGO6G12_ADC_MODE_A;

        // ADC sample value range -32768 .. 32767
        // averaging data saturates at +/- 2^21 - 1
        config->output_mode = NDIGO6G12_OUTPUT_MODE_SIGNED32;

        // enable ADC channel A and trigger on the falling edge of TRG input
        // shift baseline of analog inputs to +350 mV
        config->analog_offsets[0] = NDIGO6G12_DC_OFFSET_N_NIM * -1;

        // trigger on falling edge of TRG input
        config->trigger[NDIGO6G12_TRIGGER_TRG].edge = true;
```

(continues on next page)

(continued from previous page)

```
        config->trigger[NDIGO6G12_TRIGGER_TRG].rising = false;

        // set trigger level on TRG input to -350 mV
        config->tdc_trigger_offsets[4] = NDIGO6G12_DC_OFFSET_N_NIM;

        // enable channel
        config->trigger_block[0].enabled = true;
        // multiples of 32 ADC samples (5 ns recording time)
        config->trigger_block[0].length = 32764;

        // select TRG as trigger source of the channel
        config->trigger_block[0].sources = NDIGO6G12_TRIGGER_SOURCE_TRG;

        // configuration of the Averaging features
        // number of events that are averaged/summed
        config->average_configuration.iterations = AVERAGING_COUNT;

        // saturate averaging data instead of overflow
        config->average_configuration.use_saturation = true;

        // don't stop averaging if next iteration could lead to sample data overflow
        config->average_configuration.stop_on_overflow = false;
}
```

## 5.7 ndigo6g12_tdc.cpp

```
#include <stdio.h>
#include "ndigo6g12_app.h"

void Ndigo6GApp::ProcessTDCPacket(crono_packet* pkt) {
        // TDC packet timestamp relates to end of packet
        // adjust for timespan covered
        double packetTs =
                (double)(pkt->timestamp - pi->tdc_packet_timestamp_offset);

        // calculate packet timestamp in picoseconds
        packetTs *= pi->packet_ts_period;

        // packet length is number of 64-bit words of data
        // 2 TDC events are stored in each 64-bit chunk of packet data
        uint32_t tdcEventCount = pkt->length * 2;


        // event encoding:
        // Bits 31 downto 8: event timestamp in TDC bins relative to packet
        //                              timestamp
        // Bits 7 downto 4: event flags
        // Bits 3 downto 0: channel number
```

(continues on next page)

(continued from previous page)

```
        uint32_t* tdcEventData = (uint32_t*)(pkt->data);

        // each TDC packet covers up to 3 coarse TDC periods
        // the end of one period is marked by an event on channel 15
        uint32_t rolloverEra = 0;
        // print all TDC timestamps of the packet
        for (uint32_t i = 0; i < tdcEventCount; i++) {
                // TDC channel number
                // 0 - 3: LEMO inputs
                // 15: internal marker: end of current TDC time frame
                uint32_t tdcChannel = tdcEventData[i] & 0xf;
                // event flags
                uint32_t flags = (tdcEventData[i] >> 4) & 0xf;
                // 24-bit timestamp
                uint32_t event_ts = tdcEventData[i] >> 8;

                // valid input channel?
                if (tdcChannel < 4) {
                        // add accumulated rollovers since start of packet
                        event_ts += rolloverEra;
                        // calculate timestamp of TDC event in picoseconds
                        double edgeTsPs = event_ts * pi->tdc_period;
                        edgeTsPs += packetTs;
                        ProcessTDCTimestamp(tdcChannel, edgeTsPs);
                        printf("TDC event on channel %d timestamp: packet without "
                                "shift %.3f ns, "
                                "with shift %.3f ns, edge %.3f ns  \n",
                                tdcChannel, (double)(pkt->timestamp * pi->packet_ts_period) / 1000.0,
                                packetTs / 1000., edgeTsPs / 1000.);

                }

                if (tdcChannel == 14) {
                        // dummy data, can be ignored
                }

                // rollover marker
                if (tdcChannel == 15) {
                        rolloverEra += pi->tdc_rollover_period;
                }
        }
}

void Ndigo6GApp::ConfigureTDC(ndigo6g12_configuration* config) {
        // enable TDC channels
        for (int i = 0; i < NDIGO6G12_TDC_CHANNEL_COUNT; i++) {
                // for NIM pulses: trigger at -350 mV
                config->tdc_trigger_offsets[i] = NDIGO6G12_DC_OFFSET_N_NIM;
```

(continues on next page)

(continued from previous page)

```

                // enable TDC channel
                config->tdc_configuration.channel[i].enable = (tdcChannelMask & (1 << i)) != 0;

                // enable falling edge trigger as input to trigger matrix for selected
                // TDC channel
                // only required if used as trigger source for Gating, TiGer
                // or ADC trigger blocks
                config->trigger[NDIGO6G12_TRIGGER_TDC0 + i].edge = true;
                config->trigger[NDIGO6G12_TRIGGER_TDC0 + i].rising = false;
                // threshold not applicable for TDC inputs
                // trigger threshold is set via tdc_trigger_offsets[i]
                config->trigger[NDIGO6G12_TRIGGER_TDC0 + i].threshold = 0;
        }
}
```

## 5.8 delay.h

```
#include <deque>
#include <map>
#include <string>
#include <vector>
#include <float.h>

// this utility class manages the arrival of timestamps for
// a number of channels and tries to group them to a start
// signal on one channel and stop signals on the other channels

// delay status
enum DelayStatus {
    NotEnoughData, // we do not know, if the following signal has already arrived
    StopsMissing,  // some of the stops have arrived after a maximum wait time
    Complete,      // start and all expected stops were processed correctly
    StartMissing
};

class ChannelInfo {
  public:
    size_t index;
    int channel;
    std::string name;
    // contains the timestamps of pulses
    std::deque<double> timestamps;
    bool early;
    bool ok;
    bool HasData() const { return timestamps.size() > 0; }
};
```

(continues on next page)

(continued from previous page)

```cpp

// per channel output of delay measurement
class ChannelDelay {
  public:
    int channel;
    bool missing;
    bool isStart;
    std::string name;
    double delayPs;
    // number of events that were ignored because of missing start
    int ignoredCount;
};

// output of delay measurement
class Delays {
  public:
    DelayStatus status;
    std::vector<ChannelDelay> channelDelays;
    double startTimestamp;
};

// class for measurement of delays between given number of channels
class DelayMeasurement {
    std::vector<ChannelInfo> channels;
    // map from channel to the index in channels
    std::map<int, size_t> channelIndexes;
    //
    Delays delays;
    size_t startIndex;
    // maxDelay is the time that two timestamps are considered to be in the
    // same group, e.g., the maximum delay for a simple cable delay time
    double maxDelay;
    // maxWaitTime is the time to wait after a signal, to know that a following
    // signal has been received; this allows deciding if a group is complete
    double maxWaitTime;

  public:
    void Init(int startChannel, double maxDelay,
              std::map<int, std::string> channelMap) {
        channels.resize(channelMap.size());
        this->maxDelay = maxDelay;
        maxWaitTime = 10 * maxDelay;

        delays.channelDelays.resize(channelMap.size());
        size_t i = 0;
        for (auto const &e : channelMap) {
            int channel = e.first;
            std::string name = e.second;
            channels[i].index = i;
            channels[i].channel = channel;
```

(continues on next page)

(continued from previous page)

```
            channels[i].name = name;
            delays.channelDelays[i].channel = channel;
            delays.channelDelays[i].name = name;
            delays.channelDelays[i].isStart = startChannel == channel;
            channelIndexes[channel] = i;
            i++;
        }
        startIndex = channelIndexes[startChannel];
    }

    void SetMaxWaitTime(double maxWaitTime) {
        this->maxWaitTime = maxWaitTime;
    }
    // write the current timestamp in ps to the structure
    void InsertTimestamp(int channel, double timestamp) {
        size_t index = channelIndexes[channel];
        channels[index].timestamps.push_back(timestamp);
    }

    // the parameter is returned by pointer to avoid memory allocations
    Delays *MeasureDelays() {
        size_t maxSize = channels[0].timestamps.size();
        size_t earliestIndex = 0;
        double earliestTimestamp = DBL_MAX;
        double latestTimestamp = 0;
        delays.channelDelays.resize(channels.size());

        for (const ChannelInfo &ci : channels) {
            maxSize = std::max(maxSize, ci.timestamps.size());
            if (ci.timestamps.size() > 0 &&
                ci.timestamps.front() < earliestTimestamp) {
                earliestTimestamp = ci.timestamps.front();
                earliestIndex = ci.index;
            }
            if (ci.timestamps.size() > 0 &&
                ci.timestamps.back() > latestTimestamp) {
                latestTimestamp = ci.timestamps.back();
            }
            delays.channelDelays[ci.index].missing = false;
            delays.channelDelays[ci.index].ignoredCount = 0;
        }

        // process the queues if enough data is expected
        if (maxSize > 0 && latestTimestamp - earliestTimestamp > maxWaitTime) {
            int channelsTooEarly = 0;
            int channelsTooLate = 0;
            int channelsOk = 0;
            int channelsMissing = 0;
            bool startPresent = channels[startIndex].HasData();
            double startTimestamp = startPresent
```

(continues on next page)

(continued from previous page)

```
                                    ? channels[startIndex].timestamps[0]
                                    : latestTimestamp - 2 * maxDelay;

            for (ChannelInfo &ci : channels) {
                ci.early = false;
                ci.ok = false;
                if (ci.HasData()) {
                    double diffToStart = ci.timestamps[0] - startTimestamp;
                    delays.channelDelays[ci.index].delayPs = diffToStart;
                    if (diffToStart < -maxDelay) {
                        ci.early = true;
                        channelsTooEarly++;
                    } else if (diffToStart > maxDelay) {
                        channelsTooLate++;
                        delays.channelDelays[ci.index].missing = true;
                    } else {
                        ci.ok = true;
                        channelsOk++;
                    }
                } else {
                    if (latestTimestamp > startTimestamp + maxWaitTime) {
                        // if there was data it should have arrived by now
                        delays.channelDelays[ci.index].missing = true;
                        channelsMissing++;
                    }
                }
            }

            if (channelsOk + channelsTooLate + channelsMissing ==
                channels.size()) {
                // best case, every stop and start is included;
                // otherwise some channels are missing/too late
                for (ChannelInfo &ci : channels) {
                    if (ci.ok) {
                        ci.timestamps.pop_front();
                    }
                }
                delays.startTimestamp = startTimestamp;
                delays.status = Complete;
            } else if (channelsTooEarly > 0 || !startPresent) {
                // cut away
                double cutOffTimestamp = startPresent
                                        ? startTimestamp - maxDelay
                                        : latestTimestamp - maxWaitTime;

                bool removed = false;
                for (ChannelInfo &ci : channels) {
                    while (ci.timestamps.size() > 0 &&
                           ci.timestamps[0] < cutOffTimestamp) {
                        ci.timestamps.pop_front();
```

(continues on next page)

(continued from previous page)

```
                        delays.channelDelays[ci.index].ignoredCount++;
                        removed = true;
                    }
                }
                delays.startTimestamp = earliestTimestamp;

                delays.status = removed ? StartMissing : NotEnoughData;
            } else {
                delays.status = NotEnoughData;
            }
        } else {
            // else not enough data, process during next process packets
            delays.status = NotEnoughData;
        }
        return &delays;
    }

    // the parameter is passed by reference to avoid memory allocations
    void PrintDelays(Delays *delays) {
        if (delays->status == NotEnoughData) {
            return;
        }
        if (delays->status == Complete || delays->status == StopsMissing) {
            for (const ChannelDelay &cd : delays->channelDelays) {
                if (cd.isStart) {
                    printf("---\n%s: Start %.3lf ns\n", cd.name.c_str(),
                           delays->startTimestamp / 1000.);
                } else if (!cd.missing) {
                    printf("%s: Delay %.3lf ns\n", cd.name.c_str(),
                           cd.delayPs / 1000.);
                } else {
                    printf("%s: Missing\n", cd.name.c_str());
                }
            }
        }
        if (delays->status == StartMissing) {

            printf("---\n Start missing at  %.3lf ns\n",
                delays->startTimestamp / 1000.);
            for (const ChannelDelay &cd : delays->channelDelays) {
                if (cd.isStart) {
                    // ignore
                } else if (cd.ignoredCount > 0) {
                    printf("%s: Ignored %d\n", cd.name.c_str(),
                           cd.ignoredCount);
                }
            }
        }
    }
};
```

# 6 Technical Data

- Input Passband: **1 MHz** to **950 MHz**
- Power Requirements: **35 W**
- Mechanical Dimensions: **170 mm** × **106 mm** × **22 mm** (fits in one PCIe slot)
- Throughput: **5200 MByte/s** on PCIe x8

## 6.1 Digitizer Characteristics

Each board is tested against the values listed in the “Min” column. “Typical” is the mean value of the first 10 boards that were produced.

### 6.1.1 1-Channel-Mode (6.4 Gsps)

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $THD_1$ | Total Harmonic Distortion | | −67 | −56 | dB |
| $SNR_1$ | Signal-to-Noise Ratio | 53 | 54 | | dB |
| $SFDR_{incl,1}$ | Spurious Free Dynamic Range (including Harmonics) | 58 | 75 | | dB |
| $SFDR_{excl,1}$ | Spurious Free Dynamic Range (excluding Harmonics) | 71 | 75 | | dB |
| $SINAD_1$ | Signal-to-Interference Ratio including Noise and Distortion | 49 | 54 | | dB |
| $ENOB_1$ | Effective Number of Bits | 8.5 | 8.7 | | |

### 6.1.2 2-Channel-Mode (3.2 Gsps)

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $THD_2$ | Total Harmonic Distortion | | −70 | −56 | dB |
| $SNR_2$ | Signal-to-Noise Ratio | 54 | 54 | | dB |
| $SFDR_{incl,2}$ | Spurious Free Dynamic Range (including Harmonics) | 58 | 75 | | dB |
| $SFDR_{excl,2}$ | Spurious Free Dynamic Range (excluding Harmonics) | 71 | 77 | | dB |
| $SINAD_2$ | Signal-to-Interference Ratio including Noise and Distortion | 49 | 54 | | dB |
| $ENOB_2$ | Effective Number of Bits | 8.5 | 8.7 | | |

### 6.1.3 4-Channel-Mode (1.6 Gsps)

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $THD_4$ | Total Harmonic Distortion | | −68 | −56 | dB |
| $SNR_4$ | Signal-to-Noise Ratio | 53 | 55 | | dB |
| $SFDR_{incl,4}$ | Spurious Free Dynamic Range (including Harmonics) | 58 | 74 | | dB |
| $SFDR_{excl,4}$ | Spurious Free Dynamic Range (excluding Harmonics) | 71 | 75 | | dB |
| $SINAD_4$ | Signal-to-Interference Ratio including Noise and Distortion | 49 | 54 | | dB |
| $ENOB_4$ | Effective Number of Bits | 8.5 | 8.7 | | |

## 6.2 Oscillator Time Base

| Symbol | Parameter | Min | Typical | Max | ppb |
|---|---|---|---|---|---|
| $\Delta T$ | Temperature stability −20 °C to 70 °C[1] | | | 10 | ppb |
| $F_0$ | Initial calibration | | <300 | 500 | ppb |
| $\Delta F / F_1$ | Aging first year | | | 100 | ppb |
| $\Delta F / F_{20}$ | All inclusive aging 20 years | | | 1000 | ppb |
| | Warm-up[2] | | | 3 | min. |

[1] Over –40 °C to +85 °C; relative to stabilized frequency after 1 hour of continuous operation

[2] @+25 °C; within ±100 ppb of F, where F is the stabilized frequency reached after 1 hour of continuous operation

## 6.3 Electrical Characteristics

### 6.3.1 Environmental Conditions for Operation

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

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| T | ambient temperature | 5 | | 40 | °C |
| RH | relative humidity at 31°C non condensing | 20 | | 75 | % |

### 6.3.2 Environmental Conditions for Storage

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

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| T | ambient temperature | -30 | | 60 | °C |
| RH | relative humidity at 31°C non condensing | 10 | | 70 | % |

### 6.3.3 Power Supply

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $I_{3.3}$ | PCIe 3.3 V rail power consumption | | 0.42 | | W |
| $VCC_{3.3}$ | PCIe 3.3 V rail power supply | 3.1 | 3.3 | 3.6 | V |
| $I_{12}$ | PCIe 12 V rail power consumption[1] | | 31 | | W |
| $VCC_{12}$ | PCIe 12 V rail power supply[1] | 11.1 | 12 | 12.9 | V |
| $I_{aux}$ | PCIe 3.3 $V_{aux}$ rail power consumption | | 0 | | W |
| $VCC_{aux}$ | PCIe 3.3 $V_{aux}$ rail power supply | | 3.3 | | V |

**Note**

[1] The 12 V power is sourced solely from the PCIe power connector located at the rear of the board.

### 6.3.4 Analog Inputs

AC coupled single-ended analog inputs:

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $V_{p\text{-}p}$ | Peak-to-peak input voltage | | | 1 | V |
| $Z_p$ | Input impedance | | 50 | | Ω |
| $V_{offs}$ | Adjustable offset | -0.5 | | 0.5 | V |

### 6.3.5 Digital Inputs

AC coupled single-ended digital inputs:

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $V_{p-p}$ | Peak-to-peak input voltage | | | 1.3 | V |
| $Z_p$ | Input impedance | | 50 | | Ω |
| $I_{Term}$ | Termination Current | -50 | -20 | 50 | mA |
| $V_{offs}$ | Adjustable offset | -1.3 | | 1.3 | V |
| $t_{Pulse}$ | Pulse length | 2 | 5 | 200 | ns |
| $t_{Rise:}$ | Pulse Edge 20% to 80% | | | 10 | ns |
| $t_{Fall:}$ | Pulse Edge 80% to 20% | | | 10 | ns |

### 6.3.6 Absolute Maximum Ratings

The absolute ratings are the maximum amplitude that an input pulse can safely have before the board may be damaged.

The maximum voltage of any input voltage may not exceed the values given by $V_{max}$.

The voltages relative to a constant DC offset (i.e., the pulse "height") may not exceed the values given by $V_{AC,max}$.

#### *Analog Inputs*

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $V_{max}$ | Maximum input voltage | -25 | | 25 | V |
| $V_{AC,max}$ | Maximum pulse height relative to DC offset | -1.9 | | 2.0 | V |

#### *Digital Inputs*

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $V_{max}$ | Maximum input voltage | -16 | | 16 | V |
| $V_{AC,max}$ | Maximum pulse height relative to DC offset | -5 | | 5 | V |

## 6.4 Information Required by DIN EN 61010-1

### 6.4.1 Manufacturer

The Ndigo6G is a product of:

cronologic GmbH & Co. KG
Jahnstraße 49

60318 Frankfurt

HRA 42869 beim Amtsgericht Frankfurt/M
VAT-ID: DE235184378

### 6.4.2 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 a Ndigo6G, a main board, a case, application software and possible additional electronics to attach the system to some type of detector. They might also be integrated with the detector.

The Ndigo6G 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 PC 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 with regard to user safety and electromagnetic interference. Compliance was only tested for attached cables shorter than 3 m.

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

### 6.4.3 Environmental Conditions

See Section 6.3.1 and Section 6.3.3.

### 6.4.4 Inputs

All inputs are AC coupled. The inputs have very high input bandwidth requirements and therefore there are no circuits that provide overvoltage protection for these signals.

> **Danger**
>
> Applying high voltage on the inputs relative to the slot cover can result in permanent damage to the board. See Section 6.3.6 for the maximum ratings of the inputs.

### 6.4.5 Recycling

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

The Ndigo6G-12 belongs to **category 9**, **“Überwachungs und Kontrollinstrumente für ausschließlich gewerbliche Nutzung”.** The last owner of an Ndigo6G-12 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 in Section 6.4.1.

### 6.4.6 Export Control

The Ndigo6G product line is a dual-use item under Council Regulation (EC) No 428/2009 of 5 May 2009 in section **3A002h.** Similar regulations exist in many countries outside Europe.

Regardless of the fact that we at cronologic exclude the use of our products for military purposes, **the laws of the EU and many other countries restrict exports** of dual-use items. Since we have to apply for a **General Export Permit** for these countries, delivery processes may be delayed or delivery to certain countries may become impossible.

For the application of this export license we need the following documents from you:

- Exporter declaration
- Company profile
- Import license (country dependent)

There are countries for which a **General Export License** can be used for the export of dual-use goods. In this case we need the corresponding documents from you and there will be no further delay. Included countries are:

- Australia
- Japan
- Canada
- Liechtenstein
- New Zealand
- Norway
- Switzerland
- Singapore
- USA

Before re-exporting an Ndigo6G or any product containing an Ndigo6G as a component, please check you local regulations whether an export permit is required.

It is not permitted to export an Ndigo6G to the Russian Federation or the Republic of Belarus.

# 7 Revision History

## 7.1 Firmware

**1.25086 — 2025-04-03**

Bugfix: Removed trigger dead time

**1.24120 — 2024-04-30**

Improved ADC/TDC synchronization

Added sample averaging modes AA/DD, AAAA/DDDD, and AADD

TiGer Updates

Internal optimizations

Bug fixes

**5493 — 2023-10-30**

Fixed bug related to level triggering

Fixed first packet being empty

Minor bug fixes

**5467 — 2023-05-05**

PCIe optimizations

Minor bug fixes

## 7.2 Driver

**2.2.0 — 2025-04-03**

Bugfix: Removed trigger dead time

Bugfix: Fixed `NDIGO6G12_MAX_PRECURSOR` for *Averaging Mode*

**2.0.1 — 2024-07-17**

Extensive revision of the application programming interface

Improved linux support

Improved documentation

Improved TDC and ADC synchronization

**1.5.4 — 2024-07-13**

Fixed 2 channel handling with trigger from opposite channel (trigger A on channel D)

Fixed timestamp uncertainty in lower bits

**1.5.3 — 2024-07-07**

Dynamic reconfiguration with .cronorom support

**1.4.5 — 2023-01-23**

Crono kernel driver update to v1.4.2

Added support for revision 3 boards

Minor bug fixes

Support for 32-bit OS discontinued

**1.4.0 — 2022-08-18**

Added support for external 10 MHz reference on slot bracket

**1.3.0 — 2022-05-25**

Added support for Averager

## 7.3 User Guide

**1.4.0 — 2025-06-12**

Updated documentation of *ndigo6g12_configuration::output_mode*

Updated *ndigo6g12_param_info::adc_sample_delay* docstring

Documented possible baseline drift due to the AC coupling

Updated various figures to also work in dark mode

Improved Averager documentation

Revised documentation of *crono_packet*

**1.3.0 — 2025-04-10**

Documented firmware update procedure

Documented TDC calibration procedure

**1.2.0 — 2025-04-02**

Documented minimal packet length

Moved alert and device-state defines to corresponding documentation section

Updated Erratum

**1.1.0 — 2025-01-14**

Added Section 6.3.6

**1.0.1 — 2024-10-22**

Improved Figure 1.1

**1.0.0 — 2024-10-17**

Added digitizer characteristics

Added chapter on TiGer

Added *Erratum*

Fixed gating documentation

Many corrections

**0.2.1 — 2024-10-01**

Corrections in Export Control

**0.2.0 — 2024-10-01**

Added gating examples

Updated Export Control

**0.1.4 — 2024-08-06**

Added figures for the *Trigger Matrix* and *Gating Blocks*.

**0.1.3 — 2024-08-01**

Added documentation for clock connections

Added link to current user guide example code

Removed clutter from the APIs “ON THIS PAGE” sidebar

Updated C++ example

General improvements

**0.1.2 — 2024-07-17**

Renamed FPGA0/1 to TRG/GATE

Restructured API documentation

Expanded documentation on Packet Format

**0.1.1 — 2024-07-16**

Corrected values in introduction

Improved phrasing throughout

**0.1.0 — 2024-07-11**

Initial release

# 8 Erratum

Up to firmware revision 1.24120, the retrigger feature of the *gating blocks* does not behave as intended. Instead of a gate being only extended by a retrigger event, the state of the gate is reset to inactive.

Up to firmware revision 1.24120 and in 4-channel mode, `ndigo6g12_single_shot()` only works properly if `ndigo6g12_trigger_block::multi_shot_count == 1`.

Up to firmware revision 1.24120, a deadtime of up to 10 ns after a packet can occur.