TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ3 Series Digitizers

User Guide

Author(s): Teledyne SP Devices

Document ID: 21-2539
Classification: Public
Revision: 2023.3
Date: 2023-11-21

Contents

Document History

Section	Description	
Revision 2023.3		2023-11-21
2, 15.4, A	Document input routing parameters and allowed input routing configurations.	
9, 9.1	Document the data acquisition process for records with dynamic length.	
9.2	Document the mechanism to extend the digitizer's rearm time.	
9.5.1, 10.5.5	Document zero length records.	
10	Document the data transfer process for records with dynamic length.	
10.5.2	Document dynamically allocated record buffers.	
10.5.3	Update the flowchart for the data readout process.	
10.5.4	Document status events from the data readout process.	
10.5.6, 10.5.7	Document the new behavior for incomplete records.	
10.6, 10.6.3	Document the new overflow behavior *continue on overflow*.	
10.7	Document the new eject mechanism for partially filled transfer buffers.	
5.7	Document the FWPD firmware and its signal processing capabilities.	
Revision 2023.2		2023-05-02
Revision 2023.1		2023-01-26
Revision D		2022-04-08
Revision C		2021-06-21
Revision B		2021-02-12
Revision A		2021-02-08

1 Introduction

This document is the user guide for ADQ3 series digitizers running the standard data acquisition firmware (FWDAQ), the advanced time-domain firmware (FWATD) or the pulse detection firmware (FWPD).

Important

This document is only valid for the following digitizer models:

- ADQ30-PCIe
- ADQ32-PCIe
- ADQ33-PCIe
- ADQ36-PXIe

Release 2023.3

This document describes the state of ADQ3 series digitizers with firmware and software artifacts from release **2023.3**. Unless otherwise stated, this document is also valid for any subsequent *patch release*. These append an additional number at the end of the release label. For example, 2023.1**.1** would be the first patch release of the major release 2023.1.

1.1 Overview

Fig. 1 presents a block diagram of the functional decomposition of an ADQ3 series digitizer. Refer to the following sections for details on the respective components:

- Section 2 presents the analog front-end and its functions.
- Section 3 presents the analog-to-digital converter.
- Section 4 presents the clocking system.
- Section 5 presents the signal processing modules.
- Section 6 presents the event source system.
- Section 7 presents the function modules.
- Section 8 presents the ports.
- Section 9 presents the data acquisition process.
- Section 10 presents the data transfer and data readout processes.
- Section 11 presents the test pattern generator.
- Section 12 presents the system manager.
- Section 13 presents the front panel LEDs and their function.

- Section 14 presents the nonvolatile storage capabilities.
- Section 15 presents the application programming interface.
- Section 16 presents the Python wrapper for the application programming interface.
- Appendix A presents the API reference documentation.

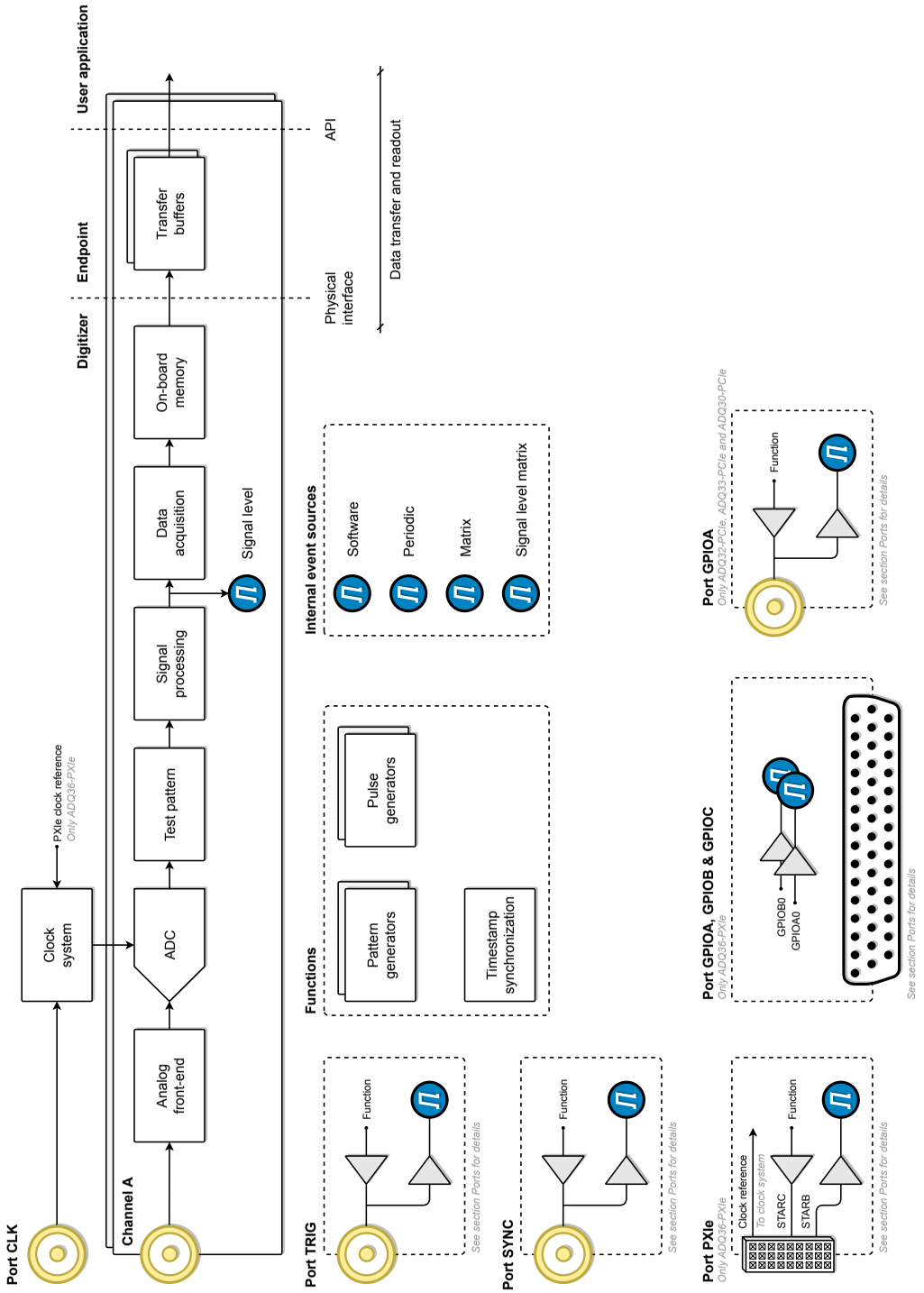


Figure 1: A block diagram presenting an overview of an ADQ3 series digitizer. The symbol ⎍ is used to indicate an event source (see Section 6). The available ports depends on the digitizer model, refer to Section 8 for more information.

1.2 How to Read This Document

This document is not intended to be read from cover to cover. It is a reference manual for the full digitizer system and should be used to learn about the many available features as needed. Not all use cases will utilize every feature offered by the digitizer. However, there are a few must-read sections:

- The event source system (Section 6).
- The data acquisition process (Section 9).
- The data transfer and readout processes (Section 10).
- The application programming interface (Section 15).

Text appearing in this color represents hyperlinks leading to another location in the document. Hyperlinks are extensively used to make it easier to navigate the content. Thus, using a PDF reader is more effective than reading this document in printed form.

The digitizer's behavior is controlled via *parameters*. These appear as hyperlinks typeset with a typewriter font, e.g. `record_length`, and are interwoven with the text. The technical details of how to *apply* a specific value to a parameter is described in Section 15.5, which outlines the configuration phase.

1.3 Firmware Types

The ADQ3 series digitizers support several different firmware types. Apart from the standard data acquisition firmware, their purpose is to specialize the digitizer to tackle domain-specific acquisition problems with greater efficiency—trading off generic acquisition properties for significant improvements in other areas.

Some information (often entire sections) in this user guide will only concern digitizers running a specific firmware type. Such information will be clearly marked and is an exception to the otherwise general rule that the information herein always applies. For more information on how to manage the digitizer's firmware, see Section 12.1.

1.3.1 Standard Data Acquisition Firmware (FWDAQ)

The FWDAQ firmware is the standard firmware available by default on all ADQ3 series digitizers.

1.3.2 Advanced Time-Domain Firmware (FWATD)

The FWATD firmware enables the use of the ATD signal processing module, which allows sample-by-sample accumulation of records. Section 5.6 provides details specific for this firmware.

1.3.3 Pulse Detection Firmware (FWPD)

The FWPD firmware enabled the use of the PD signal processing module, which allows real-time analysis of unipolar pulses. Section 5.7 provides details specific for this firmware.

1.4 The First Acquisition

To quickly get up and running with the digitizer, follow the steps outlined below:

1. Install the SDK for the host computer's platform by following the instructions in Section 15.1.
2. With the host computer powered off, install the digitizer and connect the power cable. When the system is powered on, the STAT LED should be lit with a constant green color (see Section 13.1).
3. Locate the software example `data_readout` and follow the instructions in the README to compile. See Section 15.2 for details on how to locate this software example. Do not make any changes to the example code for the first acquisition. This example follows the flow documented in Section 10.5.3.
4. Run the example application and verify that data is acquired.

1.5 Definitions and Abbreviations

Table 1 lists the definitions and abbreviations used in this document.

Table 1: Definitions and abbreviations used in this document.

Item	Description
ADC	Analog-to-digital converter
AFE	Analog front-end
API	Application programming interface
ARR	Accumulation result record (only FWATD firmware)
DC	Direct current
DMA	Direct memory access
FFC	Flexible flat cable
FFI	Foreign function interface
FIR	Finite impulse response
FWATD	Firmware featuring hardware accelerated accumulation of records
FWDAQ	Standard data acquisition firmware
FWPD	Firmware featuring real-time analysis of unipolar pulses
GPIO	General purpose input/output
GSPS	Gigasamples per second
FIR	Finite Impulse Response
Horizontal offset	The offset (in samples) between the trigger event and the first sample in the record.
I/O	Input/output
LED	Light-emitting diode
MiB	Mebibyte, i.e. $1024 \cdot 1024$ bytes.
MSB	Most significant bit
MSPS	Megasamples per second
PCIe	Peripheral component interconnect express
PDF	Portable document format
Physical interface	The device-to-host interface, e.g. PCIe.
RAM	Random access memory
Record	A dataset, usually a continuous slice of ADC samples.
SDK	Software development kit. Includes the function library, header files, supporting software tools, examples and documentation.
SSD	Solid-state drive
Trigger event	The event which triggers a record.
Infinite acquisition	A never-ending, or for practical purposes “infinite” acquisition. An acquisition can be infinite both in terms of the number of records to acquire, or in terms of the length of a record.

2 Analog Front-End

The input signal enters the digitizer hardware through one of the input connectors, passing through the *analog front-end* (AFE) before finally reaching the analog-to-digital converter (ADC). Most of the properties of the AFE are static and cannot be changed by the user during operation. For details regarding properties such as bandwidth, refer to the product datasheet [1] [2] [3] [4].

The naming convention for the analog input channels is to use letters: channel A, channel B and so on. From a programming perspective, the API uses integers starting from zero to index the available analog channels. Some digitizers support several *channel configurations* on the same hardware (e.g. `ADQ32-1CH` and `ADQ32-2CH`). This affects the *number of channels* and their *base sampling rate*. Table 2 lists the mapping between the digitizer's physical AFE channels and the index used by the API for all channel configurations.

Table 2: Mapping for the channel configurations of ADQ3 series digitizers.

AFE channel label	Default API channel index	Alternate input routing
`ADQ30-1CH`		
A	0	*N/A*
`ADQ32-2CH`, `ADQ33-2CH`		
A	0	1
B	1	0
`ADQ32-1CH`		
A	0	*Unused*
B	*Unused*	0
`ADQ36-4CH`		
A	0	1
B	1	0
C	2	3
D	3	2
`ADQ36-2CH`		
A	*Unused*	0
B	0	*Unused*
C	*Unused*	1
D	1	*Unused*

Note

The channel configuration is controlled via the digitizer's firmware. Refer to Section 12.1.1 for more information.

Note

Accessing the alternate input routing is done by modifying the `ADQInputRoutingParameters`. This is considered an advanced feature and can be safely ignored unless required by the use case. Analog performance metrics are not guaranteed to be maintained when using the alternate input routing configurations. Refer to Section 15.4 for more information.

2.1 Input Range

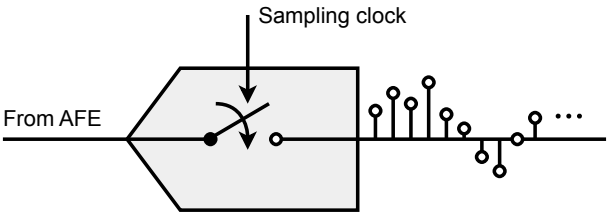
The input range determines which voltage range will map to the full scale range of the digitized samples, see (1) in Section 3 for a conversion formula. The input range is fixed on all ADQ3 series digitizers and cannot be modified. The current input range of the digitizer can be read programmatically from the channel parameter `input_range` in `ADQAnalogFrontendParameters`.

2.2 Variable DC Offset

The DC offset is a property of the AFE that may be modified by the user. By default, the value is set to zero, but may be changed to any voltage within the input range of the digitizer to inject a constant DC level into the signal path. This effectively shifts the baseline in the acquired data. Modify the channel parameter `dc_offset` to set the desired DC offset.

3 ADC

The signal from the analog front-end is converted to digital values via the ADC, at a sampling rate determined by the clock system configuration (see Section 4).



Although the digitization is performed by an ADC with 12-bit resolution, the firmware of the digitizer extends the values to 16-bit signed integers. The 12-bit ADC data is aligned to the most significant bit in the extension to 16 bits (MSB aligned).

Important

Even though the ADC data width may be lower than that of the extended data width, signal processing modules in the data path will utilize the full bit range for calculations. Therefore, ignoring the least significant bits in the output can result in a loss of accuracy.

The lower bits should not be truncated in an attempt to match the ADC bit resolution. Signal processing modules will utilize the full bit range and ignoring the least significant bits can result in a loss of accuracy. Some specific firmware types may modify this data alignment further (such as the FWATD firmware, see Section 5.6.2). Regardless of firmware type, the alignment can be read through the `code_normalization` constant parameter, which will have the value 2^{16} for the standard firmware. To convert a digitized sample to a voltage, calculate

$$x_{millivolts} = \frac{x_{codes}}{\texttt{code\_normalization}} \cdot \texttt{input\_range} - \texttt{dc\_offset} \tag{1}$$

where x_{codes} is a sample output by the digitizer and $x_{millivolts}$ is the corresponding value in millivolts. The conversion is affected by the digitizer's input range and DC offset, whose values can be read programmatically via the channel-specific analog front-end parameters `input_range` and `dc_offset`. The value of `dc_offset` defaults to zero, but can be manually changed by the user (see Section 2.2).

Values that cannot be represented in the available range will be saturated to either the minimum or the maximum representable value (whichever is closest) and the overrange bit (`ADQ_RECORD_STATUS_OVERRANGE`) will be set in the header field `record_status`.

Example

ADQ32 has an input range of 500 millivolts peak-to-peak (mVpp). A DC offset of -100 mV has been applied via the AFE. If a digitized sample has a value of 25000 codes, the corresponding voltage at the input is

$$\frac{25000}{2^{16}} \cdot 500 - (-100) \approx 290\ mV.$$

4 Clock System

The analog-to-digital conversion relies on a *sampling clock* to decide when to sample the input signal. The sampling clock can be generated in several ways, depending on the use case and the requirements. The clocking architecture consists of two parts: the high speed sampling clock generation, and the reference clock.

Important

Reconfiguring the clock system parameters will reset parts of the data path and temporarily disrupt the clock, and should be considered part of the *device initialization*. See Section 15.4.

4.1 Sampling Clock Generation

The digitizer requires a high speed sampling clock to set the rate at which the ADC digitizes the input signal. By default, this clock is created by taking a reference clock (Section 4.2) and multiplying its `reference_frequency` up to the `sampling_frequency` using a phase-locked loop (PLL). The digitizer has a limited range of sampling frequencies at which this can be done, as specified in the product datasheet [1] [2] [3] [4].

If full control over the sampling frequency is needed, it is possible to disable the digitizer's internal clock generation and provide the full sampling clock via the CLK port directly. This is accomplished by setting the `clock_generator` to `ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK` and specifying the sampling frequency via the `sampling_frequency` parameter. This value may or may not be equal to the frequency of the input clock signal, depending on the firmware:

- `ADQ30, ADQ33`
  - The `sampling_frequency` should be set to *the same* frequency as the input clock signal, and only a single frequency of 1.0 GHz is supported.
- `ADQ32`
  - `2CH-FWDAQ`, `2CH-FWATD`: The `sampling_frequency` should be set to *the same* frequency as the input clock signal, with a maximum value of 2.5 GHz.
  - `1CH-FWDAQ`, `1CH-FWATD`: The `sampling_frequency` should be set to *twice* the frequency of the input clock signal, with a maximum value of 5.0 GHz.
- `ADQ36`
  - `4CH-FWDAQ`: The `sampling_frequency` should be set to *the same* frequency as the input clock signal, with a maximum value of 2.5 GHz.
  - `2CH-FWDAQ`: The `sampling_frequency` should be set to *twice* the frequency of the input clock signal, with a maximum value of 5.0 GHz.

4.2 Reference Clock

The reference clock acts a low frequency reference point and is primarily used as a method to synchronize multiple instruments to a common time base. By default, the digitizer uses its own internal 10 MHz

reference clock as the `reference_source`. Alternatively, the digitizer can lock onto an external reference clock provided through either

- the CLK port, by setting the `reference_source` to `ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK`; or
- the PXIe backplane, by setting the `reference_source` to `ADQ_REFERENCE_CLOCK_SOURCE_PXIE_10M`.

The PXIe backplane reference clock used by the digitizer is created through combining the standard-defined 100 MHz reference clock (`PXIe_CLK100`) with the `PXIe_SYNC100` signal. This process yields a low jitter 10 MHz clock signal. It is not possible to use the standard-defined 10 MHz reference clock (`PXI_CLK10`).

Note

The PXIe backplane reference clock is only available for digitizers in the PXIe form factor.

It is also possible to output the digitizer's reference clock to other instruments via the CLK port. See Section 8 for details.

When providing an external reference clock, the digitizer can optionally process the signal before it is passed on to the sampling clock generation stage:

Low jitter mode `low_jitter_mode_enabled`

When the low jitter mode is enabled, an extra PLL stage is added to the reference clock path, prior to the sampling clock generator, which locks an internal 10 MHz oscillator to the reference clock, and then uses that oscillator as the reference for the sampling clock generation. This can potentially clean excess clock jitter from the reference, with the added constraint that the reference must be an exact multiple of 10 MHz.

Delay adjustment `delay_adjustment_enabled`

Enabling delay adjustment connects a delay line to the reference clock path with a programmable `delay_adjustment` value, which allows precise tuning of the phase of the reference clock. This can be useful when building multi-digitizer systems with synchronization requirements. For example, the daisy chain trigger mechanism (Section 7.4) can utilize this adjustment to great effect. Refer to the product datasheet for exact specifications on delay adjustment range [1] [2] [3] [4].

5 Signal Processing

This section describes the *signal processing* modules available in the digitizer. The purpose of these modules is to manipulate the continuous ADC data stream to achieve some end goal. Examples include

- vertical adjustment, in the form of digital gain and offset compensation (Section 5.1),
- sample skipping to reduce the effective data rate (Section 5.2); and
- baseline stabilization (Section 5.3).
- programmable FIR filtering (Section 5.4)

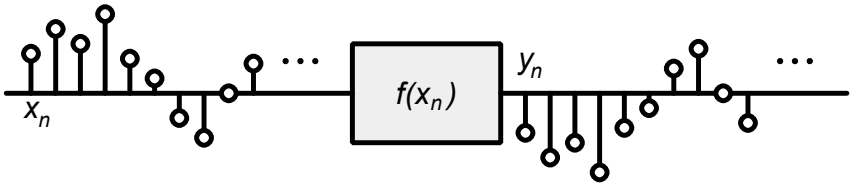


Figure 2: A typical signal processing module creates a new stream of data by applying some fixed operation to its input data stream.

5.1 Digital Gain and Offset

Digital gain and offset is a processing step that is always active and precedes all other signal processing. The module enables adjustment of the digitized samples of each channel with a `gain` value and an `offset` value as

$$y_n = \left(x_n \cdot \frac{\texttt{gain}}{\texttt{ADQ\_UNITY\_GAIN}} \right) + \texttt{offset} \tag{2}$$

where y_n is the output sample and x_n the input sample at time instance n.

Note

It is possible to specify a `gain` and/or `offset` that results in a sample value that cannot be represented in the available range. In this case, the value will be saturated to either the minimum or the maximum representable value (depending on the type of overflow) and the overrange bit (`ADQ_RECORD_STATUS_OVERRANGE`) will be set in the header field `record_status`.

5.2 Sample Skip

Sample skip is a processing step that performs data rate reduction by discarding samples. The ratio between the output data rate and the input data rate is known as the *skip factor*. This value is individually configurable for each channel via the parameter `skip_factor`. Sample skip can be a useful tool when there is a need to reduce the data rate to match the throughput of some other system component. For example, continuously writing data to a disk drive with limited write speed.

Example

An ADQ32 with a sampling rate of 2500 MHz has a `skip_factor` of 10 applied to channel A. For every 10 samples, one will be kept and 9 will be discarded. The new effective sampling rate is

$$\frac{2500\ \text{MHz}}{10} = 250\ \text{MHz}.$$

Note

The sample skip module does not low-pass filter the input data. High frequency noise will be aliased to lower frequencies when the samples are discarded. It is possible to use the FIR filter signal processing module to add some anti-alias filtering, see Section 5.4.

When synchronizing multiple digitizers with sample skip enabled, it may be useful to have the phase of the sample skipping cycles synchronized across the digitizers so that they all sample at the same time. To facilitate this, sample skip will reset its state when a timestamp synchronization is performed (see Section 7.3)

Note

By default, the `skip_factor` is set to 1, which effectively disables the sample skip module.

5.3 Digital Baseline Stabilization (DBS)

Digital baseline stabilization (DBS) is a digital algorithm that keeps the average DC value of the digitized input signal at a target `level`. It is capable of removing baseline drift from effects such as temperature changes without affecting the rest of the input signal.

DBS is especially useful in applications with pulsed input data, where the objective is often to measure pulse amplitudes relative to a baseline. By stabilizing the baseline to a preset value, such relative measurements are made easier.

The algorithm works by setting an `upper_saturation_level` and `lower_saturation_level`. These are thresholds that are relative to the baseline. Whenever the signal goes outside these thresholds, e.g. during a pulse, DBS will stop baseline estimation. As soon as the signal returns to within the thresholds, the baseline estimation resumes.

A prerequisite to using DBS successfully is that the baseline must be present in the input signal often enough, and for long enough that the algorithm can keep an accurate estimate of it. If the signal goes through a period of high activity (long segments of non-baseline content) and settles on a new baseline outside the saturation levels, baseline tracking will not resume.

Note

By default, DBS is inactive with `enabled` set to 0.

Note

A common use case in pulsed applications is measurement of unipolar pulses, where the pulses are always either positive or negative. In these situations, it is beneficial to let the baseline sit at a high offset for negative pulses (and vice versa for positive pulses) to maximize the dynamic range. To achieve this, DBS should be used in combination with the variable DC offset of the analog front-end (see Section 2). Note that it is not enough to only set the DBS target level to a high offset, as DBS adjustment is done in the digital domain. The AFE needs to shift the baseline to the target level as well.

5.4 FIR Filter

The finite impulse response (FIR) filter is a processing step which convolves the sample data with a filter impulse response. The coefficients of the impulse response can be programmed by the user to achieve different types of frequency characteristics.

The filter in the data path is a *linear-phase* filter, which means that the impulse response is symmetric. When programming the filter coefficients, only one side of the impulse response is configured, and the other side is mirrored automatically. The filter `order` is limited and can be read via the constant parameters.

Example

For a filter with `order` N, the impulse response has a total length of $N + 1$ taps. Entry 0 in the `coefficient` array corresponds to the outermost tap of the filter response, and entry $N/2$ corresponds to the center tap.

The filter coefficients are represented as fixed point values in the digitizer firmware and the specific format can be read via the constant parameters `coefficient_bits` and `coefficient_fractional_bits`. There are two ways to specify these coefficients, either

- directly as fixed point values, via the `coefficient_fixed_point` array; or
- as IEEE-754 double-precision floating point values, via the `coefficient` array.

Which method to use is specified by the `format` parameter. In the floating point case, the coefficients are subjected to rounding to convert the values into the firmware's fixed point representation. The tie-break rule used for this rounding can be specified via the `rounding_method` parameter.

Note

By default, the center tap of the filter is set to 1 and other taps are set to 0 for a flat frequency characteristic.

5.4.1 Filter Design Example

In general, filter design is outside the scope of this document and must be handled by the user. The Python package *Scipy* has several FIR filter design methods such as the *remez* and *firwin* algorithms that are easy to use. A design example for a low-pass filter is provided below.

```
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

sample_rate = 2500
filter_order = 16
passband_edge = 500
stopband_edge = 800

coefficient_fractional_bits = 14

taps = signal.remez(
    filter_order + 1,
    [0, passband_edge, stopband_edge, sample_rate / 2],
    [1, 0],
    Hz=sample_rate,
)

taps_fixed_point = np.round(taps * 2**coefficient_fractional_bits)
```

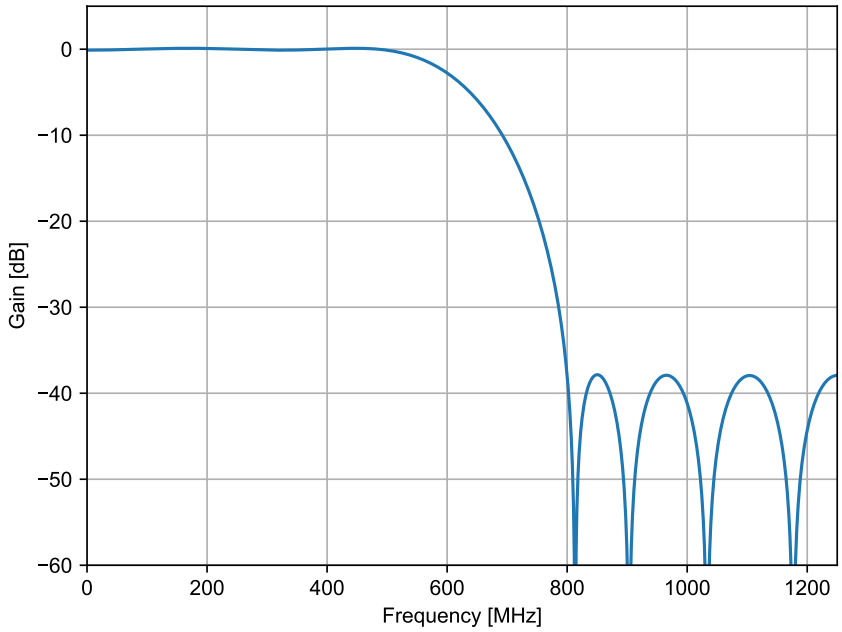


Figure 3: Example low-pass FIR filter frequency characteristic.

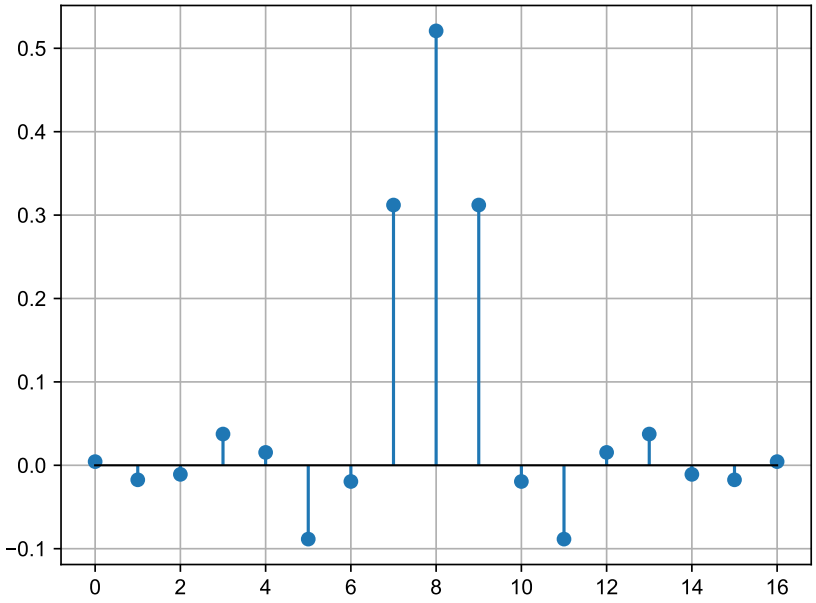


Figure 4: Example low-pass FIR filter impulse response.

5.5 PDRX

Important

PDRX is currently only supported on ADQ32 in the 2.5 GSPS mode.

The pulse dynamic range extension (PDRX) is a special operating mode of the digitizer where an analog signal is split and input to two analog channels with different gain. Each channel is digitized separately, then digitally combined into a single data stream. The combined data stream has the advantage of a higher dynamic resolution, but comes at a cost of sacrificing one (otherwise independent) analog channel. Additionally, this method is *only* effective for certain types of use cases, e.g. pulse based applications where the key parameter to extract is the amplitude ratio between pulses.

Important

PDRX is only effective in certain situations, e.g. in a pulse based application where the key parameter to extract is the amplitude difference between pulses.

Fig. 5 presents a block diagram of the parts of the digitizer specific to PDRX and should be viewed in the context of Fig. 1. In Fig. 5, the signal splitter is integrated into the digitizer hardware (see Section 5.5.1) so only one input connector is available for the two analog channels. One channel is designated as the *high gain channel* and the other is designated as the *low gain channel*. The signal in the low gain channel is subjected to an attenuation by a factor `-G` while the signal in the high gain channel is left unchanged. The two signals pass through their respective analog front-end before being digitized by the ADCs. The

data from each channel remains as two separate streams until the PDRX signal processing modules are reached. If `enabled`, the first module combines the two data streams into a single stream; otherwise, the data passes through unaltered. The rules for this combination are described in Section 5.5.2.

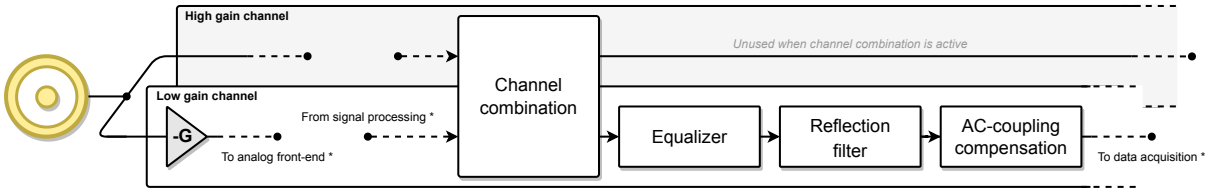


Figure 5: A block diagram showing the signal paths through the PDRX-specific hardware. Since the splitter is mounted on the digitizer, only one input connector is needed. (*) This figure should be viewed in the context of Fig. 1, which shows the full block diagram of an ADQ3 series digitizer.

It is important to note that when the PDRX operating mode is active, the combined data stream is output on the low gain channel. This is because the input range of this channel is the effective input range of the combined data stream, and of the digitizer as seen from the outside. With this convention, the calculation to translate the ADC codes into a voltage presented in (1) (Section 3) still holds true, regardless of whether the PDRX operating mode is active or not.

Important

When the PDRX operating mode is active, the combined data stream is output on the low gain channel.

5.5.1 Hardware and Firmware License

There are a few prerequisites that must be met before the digitizer can be placed in the PDRX operating mode:

- the digitizer must be fitted with an asymmetric splitter,
- the digitizer must have a valid license for PDRX; and
- the PDRX modules must be calibrated (see Section 5.5.6).

Not all ADQ3 series digitizers currently support PDRX (see Table 3) but those that do can be ordered with an *integrated* asymmetric splitter. This is specified as the `-PDRX` option when ordering.

Note

ADQ3 series digitizers that support PDRX can be ordered with the `-PDRX` option. This factory-installed option alters the analog front-end to permanently add an asymmetric splitter.

In simple terms, PDRX consists of a hardware part for *splitting* and a firmware part for *combining*. Since these actions are independent, the user can choose to construct an external asymmetric splitter (the hardware part) to evaluate PDRX on a regular ADQ3 series digitizer. To not overload the digitizer's analog front-end, the splitter must add an 18 dB attenuation for the channel designated as the *low gain channel*. The *high gain channel* and *low gain channel* designations are fixed (see Table 3) and any external asymmetric splitter must take care to attenuate the correct signal path.

Table 3: Support for PDRX across ADQ3 series digitizers. For combinations that support PDRX, the *high gain channel* and *low gain channel* designations are fixed.

Model	Firmware	Ch. A	Ch. B	Ch. C	Ch. D
ADQ30	1CH		*Unsupported*		
ADQ32	2CH	High gain	Low gain	N/A	N/A
	1CH		*Unsupported*		
ADQ33	2CH		*Unsupported*		
ADQ36	4CH		*Unsupported*		
	2CH		*Unsupported*		

The integrated asymmetric splitter will yield the best results since the reflection path between the two channels is shorter, thus lowering the probability of strong reflections being misidentified as another pulse. However, the PDRX specific hardware is permanently configured in this mode, limiting its use for more general measurement purposes.

Each channel has a PDRX-specific constant parameter set that informs the user of the digitizer's current capabilities. For example, the user can query each channel for whether or not PDRX support `is_present`. In line with the explanation in Section 5.5, this parameter will only be set for the low gain channel, i.e. the channel that receives the combined data stream—and only when supported by the correct firmware and a valid license. The associated `high_gain_channel` is also reported as a numeric index.

5.5.2 Channel Combination

The channel combination module is tasked with combining the data from the high gain channel with the data from the low gain channel to create a single data stream. Fig. 6 presents a functional block diagram. The goal is to maximize the dynamic range, so the selection mechanism always chooses samples from the high gain channel as long as the data is not clipping. The selection is carried out according to the following rules:

$$y_{out} = \begin{cases} x_{hg}, & x_{hg} \leq L, \quad \texttt{polarity} = \texttt{ADQ\_POLARITY\_POSITIVE} \\ x_{hg}, & x_{hg} \geq L, \quad \texttt{polarity} = \texttt{ADQ\_POLARITY\_NEGATIVE} \\ x_{hg}, & |x_{hg}| \leq L, \quad \texttt{polarity} = \texttt{ADQ\_POLARITY\_INVALID} \\ x_{lg}, & \text{otherwise} \end{cases} \tag{3}$$

where x_{hg} and x_{lg} are the samples from the high gain channel and low gain channel, respectively. The limit L is set automatically and is not configurable by the user. By specifying a `polarity`, the combination process can be directed to only substitute samples from the low gain channel at one edge of the input range. This is intended for a situation with unipolar pulses where the high chain channel is only expected to clip in one direction. In such cases, the dynamic range can be increased further by applying a DC offset (Section 2.2) to shift the baseline to one edge of the input range. If instead symmetric substitution is desired, use the special value `ADQ_POLARITY_INVALID`.

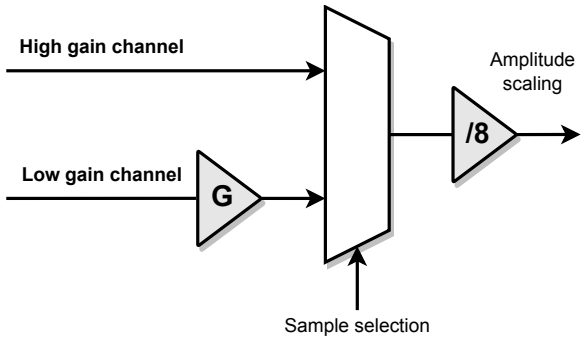


Figure 6: A functional block diagram of the PDRX channel combination process.

Before the channels are combined, the data from the low gain channel is scaled with a `gain` factor to compensate for the attenuation in the analog front-end. The value of this parameter should be determined through the calibration process described in Section 5.5.6. The combined data stream is then scaled by a fixed factor to match the range of the low gain channel. The value of the LSB of the combined channel is equal to the value of the LSB of the low gain channel.

The module is disabled by default and may be `enabled` without activating the other signal processing modules if desired, but not the other way around.

5.5.3 Equalizer

Important

The equalizer is currently not included in the digitizer firmware.

The equalizer is a signal processing module dedicated to adjusting the combined signal based on an *ideal* pulse shape. For this purpose, a general FIR filter is used, realizing the filter equation

$$y[n] = \sum_{k=0}^{N-1} c_k \cdot x[n-k], \tag{4}$$

where N is the number of coefficients (`nof_equalizer_coefficients`). Fig. 7 shows the principle of operation. The coefficients c_k are specified using the `equalizer` parameter array. The coefficients are application specific and must be calibrated by following the process outlined in Section 5.5.6.

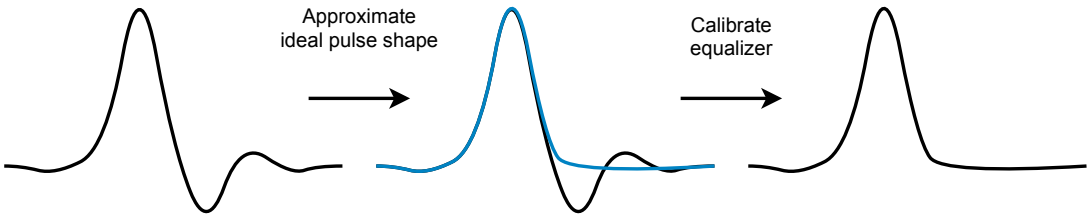


Figure 7: The principle of operation of the PDRX equalizer.

The module is disabled by default and activation requires that the channel combination is `enabled`

and that `equalizer_enabled` is set. The equalizer can *only* be used for PDRX.

5.5.4 Reflection Filter

Important

The reflection filter is currently not included in the digitizer firmware.

The reflection filter is used to reduce the impact of reflections caused by unavoidable impedance mismatch between the digitizer and external equipment. The reflections are reduced by subtracting a delayed and filtered version of the signal, implementing the expression

$$y[n] = x[n] - \sum_{k=0}^{N-1} c_k \cdot x[n-k-D], \tag{5}$$

where N is the number of coefficients (`nof_reflection_filter_coefficients`) and D is the `reflection_delay`. Fig. 8 shows the principle of operation. The coefficients, c_k are specified using the `reflection_filter` parameter array. The delay and coefficient values depend on the application and must be calibrated by following the process outlined in Section 5.5.6.

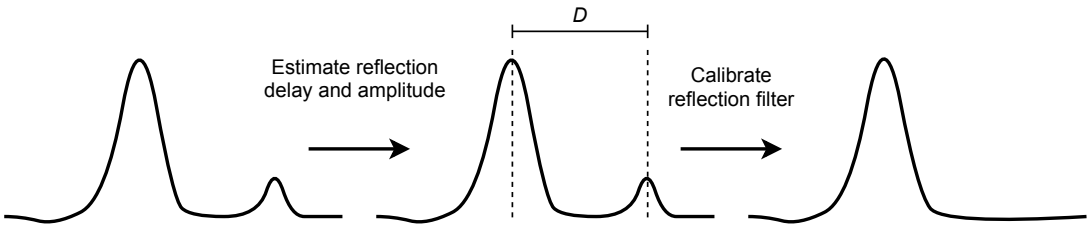


Figure 8: The principle of operation of the PDRX reflection filter.

The module is disabled by default and activation requires that the channel combination is `enabled` *and* that `reflection_filter_enabled` is set. The reflection filter can *only* be used for PDRX.

5.5.5 AC-Coupling Compensation

Important

The AC-coupling compensation is currently not included in the digitizer firmware.

While the digitizer's analog front-end is always DC-coupled, external equipment may introduce AC-coupling in the signal path. The AC-coupling compensation module is used to reduce the undesired effects in such situations. The module can *only* be used for PDRX and is *not* effective if the signal path is DC-coupled. Fig. 9 shows the principle of operation and Fig. 10 presents a functional block diagram.

Important

The AC-coupling compensation module is only effective for systems with an AC-coupled signal path. The module should *not* be used if the signal path is DC-coupled. The digitizer's analog front-end is always DC-coupled.

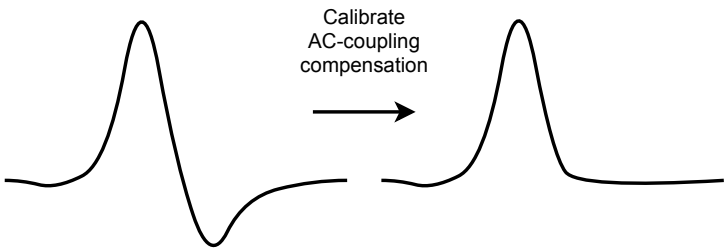


Figure 9: The principle of operation of the PDRX AC-coupling compensation.

The module consists of a high-pass IIR filter, H_{HP} followed by a baseline stabilization circuit consisting of a pulse removal module and a low-pass filter, H_{LP}, to track the baseline. The transfer function of the IIR filter is

$$H_{HP}(z) = \frac{a+1}{b+1} \cdot \frac{z-b}{z-a}, \tag{6}$$

where the pole *a* and the zero *b* is configured by the parameters `ac_compensation_filter_pole` and `ac_compensation_filter_zero`, respectively. The pulse removal circuit features two additional parameters:

- `ac_compensation_pulse_level`; and
- `ac_compensation_pulse_steepness`,

both of which are used as thresholds to condition the signal before the baseline tracking filter. The value of these parameters depend on the application and must be calibrated by following the process outlined in Section 5.5.6. The module is disabled by default and activation requires that the channel combination is `enabled` *and* that `ac_compensation_enabled` is set. The AC-coupling compensation can *only* be used for PDRX.

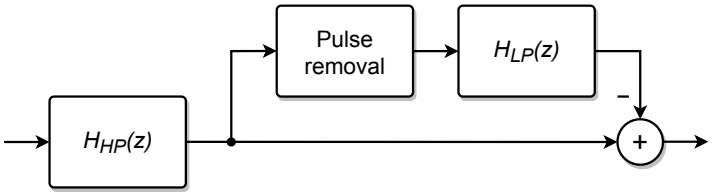


Figure 10: A functional block diagram of the PDRX AC-coupling compensation module.

5.5.6 Calibration

To be able to utilize the PDRX features in practice, the modules *must* be calibrated. At a minimum, the gain compensation factor used for the channel combination (Section 5.5.2) must be estimated. The

remaining modules: the equalizer, the reflection filter and the AC-coupling compensation only requires calibration if they are intended to be used. These modules must be calibrated with the digitizer *fully integrated* in the enclosing system since their purpose is to cancel undesired *external* effects. The calibration data is *unique* to a specific digitizer in one specific context.

Important

The equalizer, AC-coupling compensation and reflection filter all compensate for undesired external effects. Thus, the digitizer must be calibrated once it is fully integrated in the enclosing system.

Important

Calibration data is not transferrable between digitizers and systems.

To simplify the calibration process, the software development kit (Section 15.1) includes a command line application dedicated to this task. The application will target a specific digitizer, acquire data and calculate values for all the PDRX parameters. The user will need to provide seed values for a few parameters. However, most values are inferred automatically from analyzing the calibration signal.

The result is a file containing the JSON representation (Section 15.8.2) for the PDRX parameter set. The file is associated with a specific digitizer and its contents can be applied by calling `SetParameters-Filename()`. Since the file only contains the PDRX parameter set, the operation will overwrite any existing PDRX parameter values but leave all other parts of the digitizer unaffected.

Many aspects of the calibration process can be controlled via command line options. By default, the process consists of acquiring a few hundred records using a trigger signal connected to the TRIG port. The records are averaged to reduce the impact of noise, thus each record is expected to have roughly the same signal content. Additionally, the signal must meet the following specification:

- There must be at least one pulse per record.
- The baseline must have time to settle between pulses.
- None of the pulses must clip at the top or bottom of the input range of the low gain channel.
- For each pulse, there must be some samples that do not clip at the top or bottom of the input range of the high gain channel.

Important

Refer to the application's help text for more details.

Example

```
> adqcalibratepdrx --approximate-reflection-delay=30 --dc-offset=-1000
```

The approximate reflection delay has been given a seed value of 30 samples. This value is a rough estimation of the time delta between a pulse and its reflection due to impedance mismatch. The value must be provided by the user to calibrate the reflection filter. The process will use this value to carry out a search for reflections. Minor adjustments, e.g. using 31 samples or 29 samples is not expected to affect the result noticeably. Additionally, a negative DC offset is applied, allowing a better utilization of the input range for unipolar pulses. The calibration should be performed using the same DC offset value as the user application.

Example

```
> adqcalibratepdrx --ac-coupling-cutoff-frequency=1591549.4
```

In an AC-coupled system, the AC-coupling cutoff frequency (-3 dB limit) must be provided by the user. If this value is not specified, the AC-coupling compensation is disabled—which can lead to problems when calibrating the equalizer since it inherits the responsibility of smoothing out the baseline. In this example, the value is set to 1.59 MHz, corresponding to a system with an AC-coupling capacitance of 1 nF and a source and a digitizer both terminated with 50 Ω.

$$f_{3dB} = \frac{1}{2\pi RC} = \frac{1}{2\pi \cdot (50 + 50) \cdot 1 \cdot 10^{-9}} \approx 1.59 \text{ MHz}$$

Example

```
> adqcalibratepdrx --equalizer-start-offset=0 --equalizer-stop-offset=40
```

When the equalizer is calibrated, the falling edge of the pulse is replaced with that of an ideal Gaussian pulse. The replaced segment is specified in samples relative to the peak and can be controlled via the two options `--equalizer-start-offset` and `--equalizer-stop-offset`.

Example

```
> adqcalibratepdrx --width-estimation-samples=40
```

When creating the equalizer reference signal, the width of each pulse is estimated by first solving the Gaussian function individually for samples close to the peak and for each fitted curve, calculating the full width at half the maximum value (FWHM) and then taking the average of the results. In this example, the range is set to 40 samples, instructing the estimation to look at 20 samples on each side of the peak. If a pulse is wider than 40 samples, the estimation will fail to find suitable samples for the Gaussian fit. Thus, this value should be set so there are no pulses whose width (in samples) exceed this number.

5.6 ATD

Important

This section only applies to digitizers running the FWATD firmware.

The ATD signal processing module enables *hardware accelerated* accumulation of records as well as filtering of the record data based on a threshold. These features are *only* present when the digitizer is running the FWATD firmware, which is not available by default and must be purchased separately. Fig. 11 presents a block diagram of the parts of the digitizer specific to this firmware and should be viewed in the context of Fig. 1, which gives the general overview.

Important

Hardware accelerated accumulation of records is *only* available for digitizers running the FWATD firmware. This firmware is currently only supported on ADQ30, ADQ32 and ADQ33.

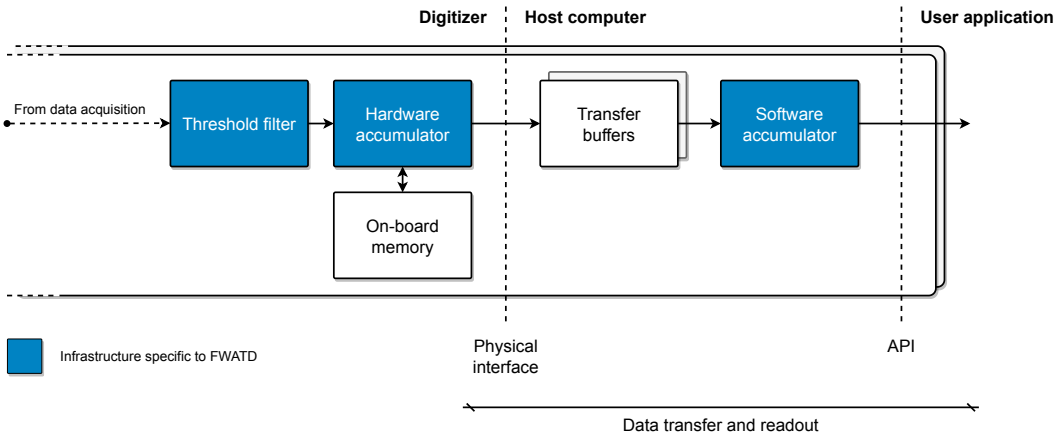


Figure 11: A block diagram of the signal processing modules unique to the FWATD firmware. This figure should be viewed in the context of Fig. 1, which presents the general overview.

Accumulator

The accumulator receives consecutive records generated by the data acquisition process (see Section 9) and accumulates them sample by sample, resulting in a single record containing the accumulated data. The accumulator resides partly in firmware and partly in software (inside the API). Refer to Section 5.6.2 for more information.

Threshold filter

The record data can be passed through a threshold filter prior to accumulation. This helps to avoid accumulating noise, and can effectively increase the dynamic range of the acquisition and help isolate rare or “weak” regions of interest within a record, e.g. an infrequently occurring pulse. Refer to Section 5.6.3 for more information.

5.6.1 FWATD Firmware and License

In addition to the FWATD firmware, the digitizer must also have a valid license for this firmware. In the event that a valid license is missing, the digitizer will fail to start and a message will be added to the trace log. For information on how to program a firmware onto the digitizer, or to switch between the available firmware images, see Section 12.1. For information on how to transfer a valid FWATD firmware license to the digitizer, see Section 12.2.

5.6.2 Accumulator

The accumulator is *partitioned*, meaning that the workload is split between the digitizer and the host computer. There are two separate 32-bit accumulators working in tandem: one implemented in the digitizer's firmware and one implemented in the API as part of the data readout process.

Important

The record accumulator is partly implemented in the data readout process (in software). Therefore, using the data transfer interface directly (Section 10.4) is *not* supported and neither is transferring data to any other endpoint than the host computer.

The number of accumulations is specified by the parameter `nof_accumulations`, and determines the number of consecutive records in time which should be accumulated and presented as a single record—the *accumulation result record* (ARR). This definition means that zero is an invalid value and "no accumulation" is specified as the value 1 (default if the FWATD firmware is running).

Important

Digitizers not running the FWATD firmware *must* keep the parameter `nof_accumulations` set to zero, which will be the default value in those cases. The easiest way to fulfill this requirement is to not modify the parameter at all.

The number of accumulations is passed through a partitioning algorithm to determine how to divide the workload between the digitizer and the host computer. The algorithm is weighted to perform as many accumulations as possible in hardware. Since the on-board DRAM is finite in size, shorter records will enable a higher number of accumulations to be performed in hardware while longer records shifts the boundary in the other direction.

The accumulator will scale down the data from its original 16-bit width prior to the hardware accumulator. This is reflected in the `code_normalization` constant parameter which has a lower value for a digitizer running the FWATD firmware compared to one running the FWDAQ firmware. This scaling is fixed and cannot be changed by the user.

An *arithmetic overflow* occurs when the accumulation process results in a value that cannot be represented as a 32-bit integer. In these situations, the value will be saturated to either the minimum or the maximum representable value (depending on the type of overflow) and the overrange bit (`ADQ_RECORD_STATUS_OVERRANGE`) will be set in the header field `record_status`.

Note

For a `code_normalization` value of 2^N, records can safely be accumulated 2^{32-N} times without any risk for arithmetic overflow. For higher accumulation values, the risk of arithmetic overflow will depend on the characteristics of the input signal.

Record Data and Metadata

The result of each completed accumulation is a single record—the accumulation result record (ARR). This resulting record is output to the user application space through the data readout process (see Section 10.5). However, the record's sample data will be in 32-bit format, with each sample holding the sum of the corresponding samples in all of the accumulated records. The size of a sample can be read programmatically from the parameter `bytes_per_sample`, as well as inferred from the value of the header field `data_format`. The accumulation result record uses the same header format as records from the standard data acquisition process (defined by `ADQGen4RecordHeader`), with some notable behavioral differences:

- Record metadata such as the `timestamp` will be taken from the *first* record that is used in the accumulation. An exception is the `record_number`, which increments by one for each consecutive ARR.
- The `firmware_specific` field will hold the number of records that were accumulated to create the accumulation result record.
- The `data_format` will be set to `ADQ_DATA_FORMAT_INT32`, indicating that the data consists of 32-bit signed integers.
- The overrange bit, defined by `ADQ_RECORD_STATUS_OVERRANGE`, in `record_status` will be set on an arithmetic overflow.

Code to Voltage Conversion

With the record accumulator enabled, the expression presented in (1) (Section 3) to convert an ADC code to voltage must be adjusted to take the number of accumulations into account:

$$x_{millivolts} = \frac{x_{codes}}{\texttt{code\_normalization} \cdot \texttt{firmware\_specific}} \cdot \texttt{input\_range} - \texttt{dc\_offset}, \tag{7}$$

where x_{codes} is a sample output by the digitizer and $x_{millivolts}$ is the corresponding value in millivolts.

Important

The `firmware_specific` field from the record header must be used in (7), rather than the value of `nof_accumulations`, since the actual number of accumulations for a given record can be lower than the desired number in the event of an overflow. See Section 5.6.6 for more information.

5.6.3 Threshold Filter

The threshold filter consists of a programmable linear-phase FIR filter and decision logic aimed to single out weak unipolar pulses from the surrounding noise. A block diagram of the threshold filter is presented in Fig. 12.

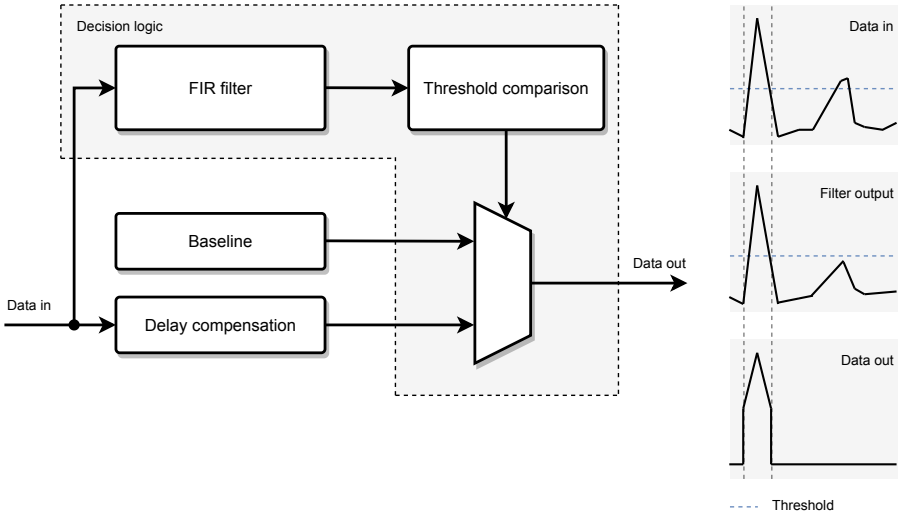


Figure 12: Block diagram of the threshold function.

The decision logic consists of a multiplexer which selects between either the incoming sample data or a programmable `baseline` value, on a sample-by-sample basis. The decision logic compares the user-defined threshold `level` to the filter output and depending on the `polarity`, determines if the sample should be replaced by the baseline value or if it should remain unchanged. By replacing undesired samples with the constant baseline value, accumulation of noise can be avoided and the dynamic range in the resulting accumulation is increased.

The threshold filter is disabled by default and can be `enabled` on a per-channel basis. This action is only allowed if the FWATD firmware is running. Otherwise, the operation will return an error.

Important

Digitizers not running the FWATD firmware *must* keep each threshold filter disabled (`enabled` set to zero). The easiest way to fulfill this requirement is not to modify the parameter at all.

Note

The threshold filter is applied prior to the scaling described in Section 5.6.2. Therefore, the threshold `level` should be set relative to the full 16-bit sample code range.

Decision Logic

The decision logic branch contains the programmable linear-phase FIR filter (symmetric impulse response). The data passed through the filter does *not* proceed to the accumulator. Instead, the filter is

used to shape the data that is compared against the threshold value to determine whether the samples should be substituted with the baseline value. This substitution occurs if the relation in (8) corresponding to the current `polarity` is met, $y(n)$ is the FIR filter output.

$$\begin{aligned} y(n) &< \texttt{level}, \quad \text{if } \texttt{polarity} \text{ is } \texttt{ADQ\_POLARITY\_POSITIVE} \\ y(n) &> \texttt{level}, \quad \text{if } \texttt{polarity} \text{ is } \texttt{ADQ\_POLARITY\_NEGATIVE} \end{aligned} \tag{8}$$

An example of this is shown in Fig. 12, where the filter has been designed to attenuate the rightmost undesired pulse. While the unfiltered data crossed the threshold level, the filter output does not, and thus the pulse is filtered out.

Example

Entry 0 in the `coefficient` array corresponds to the outermost tap of the filter response, and entry `nof_coefficients` − 1 corresponds to the center tap.

The filter has a total of `nof_coefficients` coefficients, which are represented as fixed point values in the digitizer firmware and the specific format can be read via the constant parameters `coefficient_bits` and `coefficient_fractional_bits`. There are two ways to specify these coefficients, either

- directly as fixed point values, via the `coefficient_fixed_point` array; or
- as IEEE-754 double-precision floating point values, via the `coefficient` array.

Which method to use is specified by the `format` parameter. In the floating point case, the coefficients are subjected to rounding to convert the values into the firmware's fixed point representation. The tie-break rule used for this rounding can be specified via the `rounding_method` parameter.

Note

By default, the center tap of the filter is set to 1 and other taps are set to 0 for a flat frequency characteristic.

5.6.4 Limitations

Important

The limitations imposed on the data acquisition process listed in this section are specific to the FWATD firmware and supersede any limitations (or lack thereof) mentioned in Section 9.

The ATD signal processing module requires that all channels of the digitizer generate records *synchronously*. This means that all `channel` parameters for the data acquisition process must be *identical* across all channels and that no channel can be disabled.

Additionally, the ATD signal processing module must also be able to divide the incoming record data evenly into *segments*, which results in an additional constraint on the `record_length`:

$$\texttt{record\_length} = S \cdot R \tag{9}$$

where S can be any valid value in the range specified in Table 4, and R is an integer. The calculation of

segment value S is performed automatically by the API when the `record_length` is set and will fail if no valid value for S is found.

Table 4: Granularity and minimum / maximum values for segment length S, which must evenly divide the `record_length` when the digitizer is running the FWATD firmware.

Model	Firmware	S_{min}	S_{max}	S_{step}
ADQ30	1CH	256	8192	32
ADQ32	2CH	128	4096	16
	1CH	256	8192	32
ADQ33	2CH	128	4096	16
ADQ36	4CH		*Unsupported*	
	2CH		*Unsupported*	

5.6.5 Accumulation Grid Synchronization

When `StartDataAcquisition()` is called and the digitizer transitions to the acquisition phase, trigger events will start to generate records according to the normal rules of the data acquisition process.

Let $N_A =$ `nof_accumulations` and assume that no (additional) trigger events occur during a record, i.e. that a trigger event is synonymous to a record. The accumulator will collect the first N_A records before emitting a single ARR to the user application space. Record $N_A + 1$ will be the first record in the next ARR. The grid established by the trigger events of the *first* record in each ARR is referred to as the *accumulation grid*.

For applications with constant trigger rates, the accumulation grid is well-defined. However, for some applications the trigger source can be unreliable, and generate more than N_A events in one burst, and other times fewer than N_A events. To handle these scenarios, the accumulation grid can be resynchronized by the output signal from any of the pattern generators (see Section 7.1), and thus by extension, from an external signal. Grid resynchronization is disabled by default, but can enabled by configuring one of the pattern generators and selecting it as the `accumulation_grid_synchronization_source`.

It is safe to synchronize the grid at any point during an acquisition, but the action will always cause a transient behavior in user space. This behavior is well-defined and regulated by the way the system handles an overflow. Please refer to Section 5.6.6 to understand the possible outcomes.

Important

It is not possible to set a *maximum* number of accumulations N_A, and expect any number of trigger events $< N_A$ to consistently generate an ARR by resynchronizing the accumulation grid. This is because an ARR may be discarded due to the rules of the overflow mechanism (Section 5.6.6).

Example

Fig. 13 presents a timing diagram demonstrating the accumulation grid synchronization in three different cases:

- the incoming triggers exactly match the number of accumulations (region 0),
- the incoming triggers fail to reach the number of accumulations (region 1) and
- the incoming triggers exceed the number of accumulations (region 2).

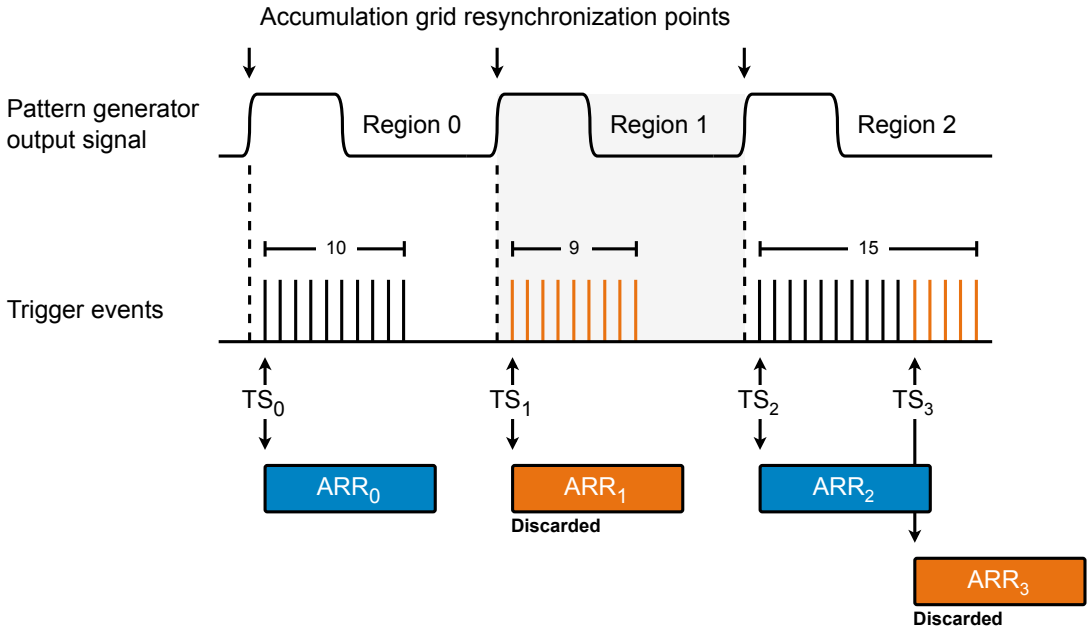


Figure 13: A timing diagram demonstrating the accumulation grid synchronization. There are three regions, each initiated by a rising edge of the output signal from the selected pattern generator. Respectively, they visualize three cases where the incoming trigger events exactly match, fail to reach or exceed the number of accumulations $N_A = 10$.

The record accumulator is configured to carry out 10 accumulations ($N_A = 10$) and the pattern generator is set up to emit a pulse with a period that covers the worst-case time range for the source to output a burst of triggers, and the accumulation grid synchronization is enabled.

Region 0

The first region starts at the first rising edge of the pattern generator's output signal. At this point, the accumulation grid is resynchronized and the first trigger event that follows marks the start of the first accumulation result record, ARR_0. The timestamp of this record, TS_0, will propagate to the user via the record `header`. The region contains exactly 10 triggers which all fall inside of the pattern generator period.

Region 1

The second region starts at the second rising edge of the pattern generator's output signal. Once again, the accumulation grid is resynchronized and the first trigger event that follows marks the start of ARR_1. The timestamp is TS_1 and taken exactly at the point where the first record in the region is triggered. However, this time only 9 trigger events fall within the region and thus

the requirement of $N_A = 10$ is not satisfied. The ARR will ultimately be discarded, but the actual decision to do so is deferred to the start of region 2, where any ongoing (incomplete) accumulation is abandoned.

Region 2

The third region starts at the third rising edge of the pattern generator's output signal. The accumulation grid is resynchronized, causing the ongoing accumulation of ARR_1 to be discarded. The first trigger event that follows marks the start of ARR_2 (timestamp TS_2). The region contains 15 triggers in total, which means that the first 10 will be used to complete ARR_2 and the one following will mark the start of ARR_3 (timestamp TS_3). Similar to what occurred in region 1, ARR_3 will be discarded at the next grid synchronization event, having only received 5 trigger events.

5.6.6 Overflow

Note

This section deals with overflows caused by a data rate imbalance, not to be confused with an *arithmetic overflow* (Section 5.6.2).

The FWATD firmware features a data discarding mechanism to deal with the many different types of overflow that may occur due to data rate imbalances. The core concept is to discard data in a well-defined manner such that the overall accumulation grid (see Section 5.6.5) is preserved and corrupted records are prevented from propagating to the user.

An overflow is caused by a stall of the data transfer interface for an extended period of time. This may in turn be caused by an imbalance between the transfer bandwidth of the device-to-host interface and the output data rate of the digitizer.

During an overflow, data collected up until the point of overflow remains intact, while incoming data is discarded in a well-defined manner. An overflow will manifest itself in two possible ways, depending on the accumulator settings and how the workload has been partitioned between the hardware and software accumulators. Both of these events are discernible to the user by reading the information available in the ARR `header`.

1. An ARR will contain fewer number of accumulated records than the defined number of accumulations. This can be detected through the `firmware_specific` field, which will contain a value lower than `nof_accumulations`.

2. One or several ARRs will be missing completely. This can be detected through the `record_number` header field, which will increment by a value higher than 1 if an accumulation has been discarded.

What is guaranteed not to happen is the corruption of data, e.g. that some regions in an ARR are the result of X accumulated record while Y is the number of accumulated records for other regions.

A different type of overflow occurs if the trigger rate is not well-matched to the record length. For example, if the trigger period is 8 μs and the record length spans 10 μs, the digitizer will still be recording data for the previous record when a new trigger event occurs. In this case, the trigger event is simply ignored, causing the effective trigger period to be 16 μs.

5.7 PD

Important

This section only applies to digitizers running the FWPD firmware.

The PD signal processing module enables *hardware accelerated* analysis of unipolar pulses, extracting key *attributes*: the peak value, its position, the area and the full width at half maximum (FWHM). These pulse attributes are transferred in a dedicated channel which can be controlled independently from their associated analog channel (also called *source channel* throughout this section). Ultimately, the FWPD firmware can be configured to *only* transfer the pulse attributes, significantly reducing the data rate to the endpoint while still analyzing data at the ADC's sampling rate.

Fig. 14 presents a block diagram of the parts of the digitizer specific to this firmware and should be viewed in the context of Fig. 1, which gives the general overview.

Important

Hardware accelerated analysis of unipolar pulses is *only* available for digitizers running the FWPD firmware. This firmware is currently only supported on ADQ30, ADQ32, ADQ33 and ADQ36.

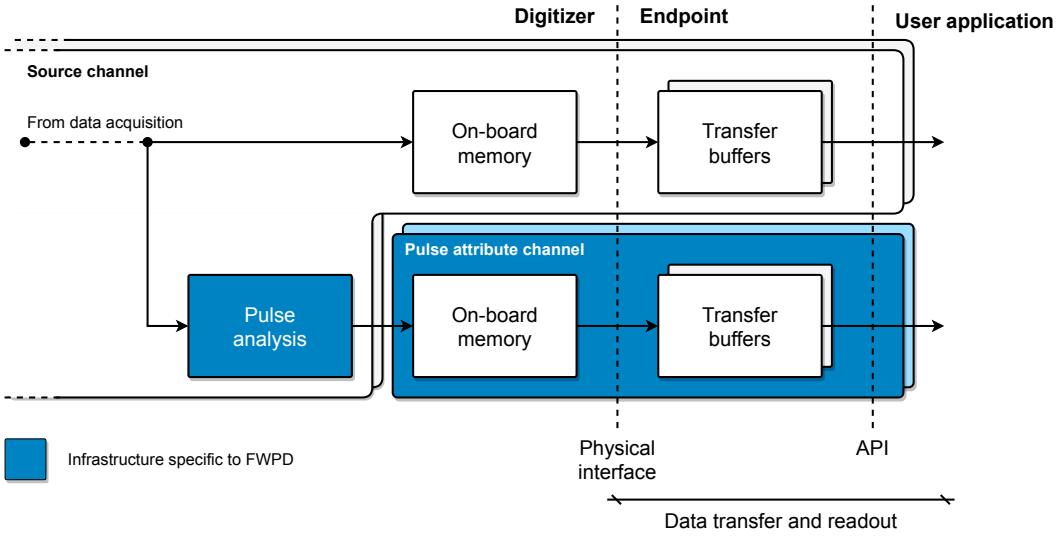


Figure 14: A block diagram of the infrastructure unique to the FWPD firmware. This figure should be viewed in the context of Fig. 1, which presents the general overview.

Each source channel has a dedicated analysis module that extracts the attributes from each pulse identified in the channel's data stream. This identification is controlled by the channel's signal level event source (Section 6.4). Thus, specifying correct values for the `level` and `arm_hysteresis` is critical for the analysis process to work as expected. A pulse is defined as the samples

- from (including) the first sample at or after the `level` crossing event at the leading edge; and
- to (excluding) the first sample after the `level` crossing event at the trailing edge.

Each analysis module outputs its data on a separate channel dedicated to pulse attribute data. From the programmer's perspective, these channels are placed *after* the source channels in terms of indexing (see Table 2 Section 2). For example, a digitizer with two source channels A and B at indexes 0 and 1 will have two additional pulse attribute channels at indexes 2 and 3, for pulse attributes from channel A and B, respectively. This brings the total number of channels involved in the data transfer and readout processes (Section 10) to twice the number of analog channels and is one of the main differences between the FWPD firmware and the standard FWDAQ firmware. This number is available as the constant parameter `nof_transfer_channels`.

5.7.1 FWPD Firmware and License

In addition to the FWPD firmware, the digitizer must also have a valid license for this firmware. In the event that a valid license is missing, the digitizer will fail to start and a message will be added to the trace log. For information on how to program a firmware onto the digitizer, or to switch between the available firmware images, see Section 12.1. For information on how to transfer a valid FWPD firmware license to the digitizer, see Section 12.2.

5.7.2 Pulse Analysis

The pulse analysis is framed by the records generated by the data acquisition process (Section 9), meaning that only pulses residing within a record from a source channel will be analyzed. A record can thus be viewed as a *detection window*. Each *source record* will generate exactly one attribute record on the corresponding pulse attribute channel. The latter will contain the analysis result for each pulse in the source record in chronological order. Since the number of pulses may vary, the pulse attribute record will have its length determined *dynamically*, requiring the data transfer process to be configured for records with dynamic length. Refer to Section 10.5.3 for more information. If the source record did not contain any pulses, a zero length record (Sections 9.5.1 and 10.5.5) is emitted.

The analysis process assumes unipolar pulse data, i.e. pulses extend in a single direction relative to the `baseline`. The `polarity` is a configurable parameter that needs to be set to either `ADQ_POLARITY_POSITIVE` or `ADQ_POLARITY_NEGATIVE` according to the use case. The `polarity` will influence the search direction for the `peak` value, as well as the way each sample contributes to the `area`. Refer to Sections 5.7.5–5.7.7 for additional details.

The `baseline` is a configurable value that needs to be set to the expected reference level for the ADC data. There is no dynamic baseline estimation built into the analysis module. If baseline drift is a problem, it is recommended to use the digital baseline stabilization module (Section 5.3) to ensure a stable reference level, and to set the `baseline` to the *same* target `level`.

Important

The analysis parameters are set for the analog channels, *not* the attribute channels.

Fig. 15 presents an example of how the pulse analysis operates in general as well as in a few corner cases. In the figure, the data acquisition process is configured to acquire records with static length triggered by an external source, e.g. `ADQ_EVENT_SOURCE_TRIG`. Pulses where both `level` crossing events reside within the source record will generate attribute data that is placed in a record emitted on the pulse

attribute channel. Attributes extracted from the pulses in the source record are placed back-to-back in the attribute record.

A pulse whose leading edge falls outside the record is ignored entirely. However, if a pulse lies partially within the source record with its trailing edge outside, an invalid analysis result is generated—as if the pulse was cut short due to reaching the maximum length limit (see Section 5.7.4).

Pulse attribute records will copy their timing information from the corresponding source record. This way, the position of the peaks on the digitizer's timing grid can be fully determined. See Section 5.7.5 for more information.

5.7.3 Data Format

The `data` in a record emitted by one of the pulse attribute channels will consist of several `ADQPulseAttributes` objects. There will be as many objects as there are pulses in the corresponding source record. The `ADQPulseAttributes` are placed back-to-back in chronological order in the memory pointed to by `data` (see Fig. 15). The timing information in the record `header` will be copied from the source record, i.e. the `timestamp` will point to the trigger event and `record_start` will indicate its relative distance to the first sample in the source record. This gives a well-defined frame of reference for the pulse attribute data and the relative timing information within, such as the `peak_position`.

Important

If a record contains pulse attribute data, the header field `data_format` will be set to `ADQ_DATA_FORMAT_PULSE_ATTRIBUTES`.

5.7.4 Limitations

Due to the nature of some of the calculated attributes, there is an upper limit to the length of a pulse that can be correctly analyzed. Above this maximum length, some attributes become *invalid* and the corresponding values reported in the `ADQPulseAttributes` object cannot be trusted. Trustworthy attribute data is signaled by `ADQ_PULSE_ATTRIBUTES_STATUS_VALID` in the `status` field, and the absence of this flag implies that one or several attributes are invalid. The maximum pulse length varies depending on on the digitizer model and the current channel configuration, see Table 5.

The highest pulse rate that can be tolerated indefinitely (the maximum *sustained rate*) also depends on the digitizer model and the current channel configuration. Rates exceeding this limit can only be tolerated for short periods (in *bursts*) before causing an overflow that forces attribute data to be discarded. This limit stems from the size imbalance between the shortest possible pulse of *one* sample and the size of the `ADQPulseAttributes` object. The highest burst rate is achieved for a sequence of one-sample pulses every other sample and can be expressed generally as

$$\texttt{sampling\_frequency} \,/\, 2.$$

This rate can be handled correctly for a sequence of 256 consecutive pulses before causing on overflow. The pulse rate must drop below the maximum sustained rate listed in Table 5 to restore normal operating conditions. The overflow status can be queried via `GetStatus()`.

Another limitation of the FWPD firmware is the fact that the write bandwidth of the on-board memory is balanced around supporting acquisitions with 100% duty cycle for *either* the source channels *or* the

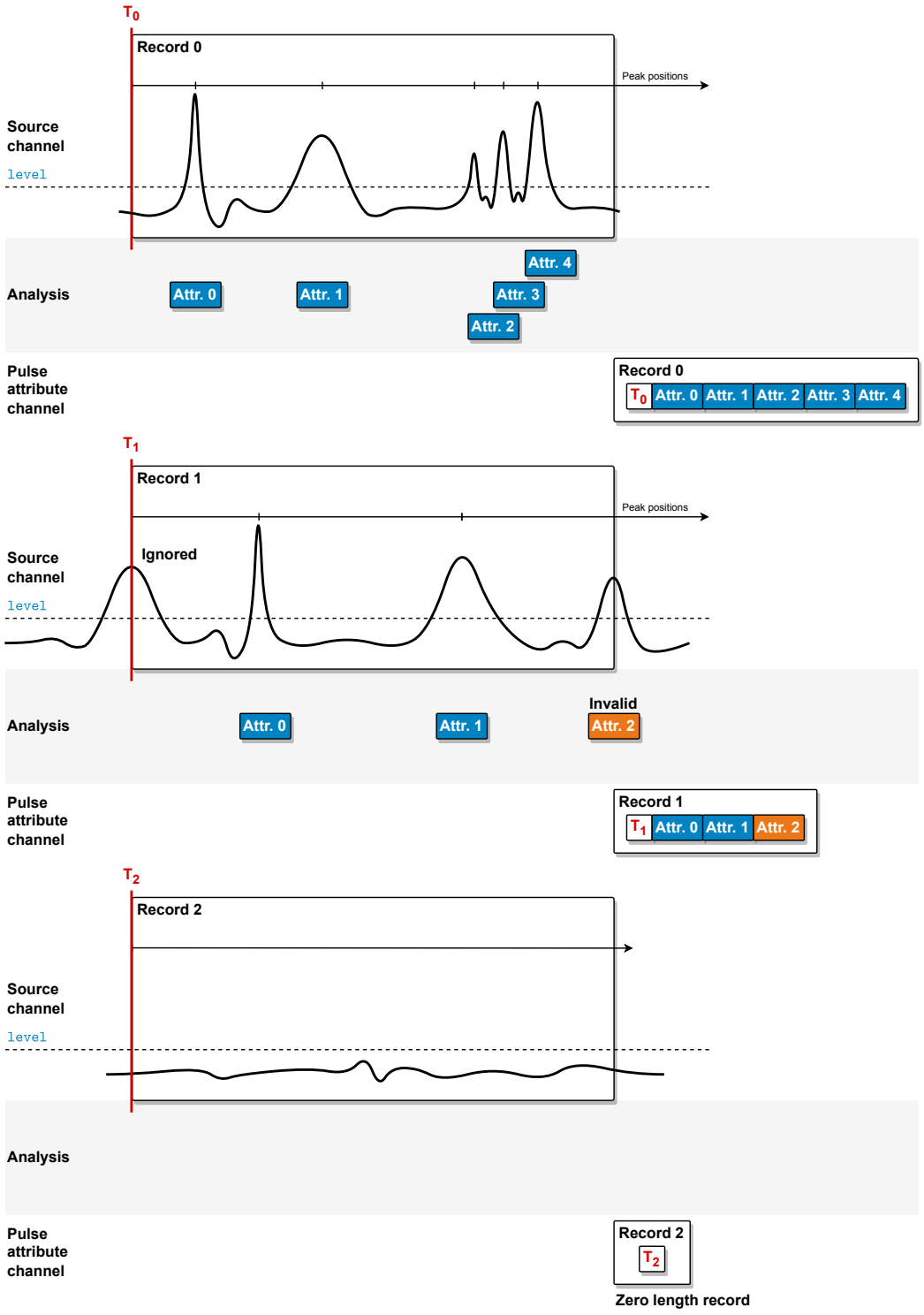


Figure 15: An example of how pulse attributes are extracted depending on the contents of a record. Pulses that do not reside fully within in the record will either be ignored or generate an invalid analysis result. A record without any pulses will generate a zero length record.

attribute channels, but not both at the same time. While transferring data from all available channels is always allowed, in doing so there is a risk of causing an overflow (Section 10.6.1) *regardless* of the current fill level of the on-board memory. This is not expected to be a problem in practice, since the point of the FWPD firmware is ultimately to only transfer attribute data. Transferring both source and attribute data is intended for debugging and verification.

Table 5: Limitations of the FWPD pulse analysis depending on the digitizer model and its channel configuration. A pulse whose length exceeds the limit will not signal `ADQ_PULSE_ATTRIBUTES_STATUS_VALID` in the `status` field. Pulse rates higher than the sustained rate can only be tolerated in short bursts before causing an overflow.

Model	**Firmware**	**Maximum length** [S]	**Maximum sustained rate** [Hz]
ADQ30	1CH	8192	`sampling_frequency` / 8
ADQ32	2CH	8192	`sampling_frequency` / 8
	1CH	16384	`sampling_frequency` / 16
ADQ33	2CH	8192	`sampling_frequency` / 8
ADQ36	4CH	8192	`sampling_frequency` / 8
	2CH	16384	`sampling_frequency` / 16

5.7.5 Peak Value and Position

The *peak* is defined as the point at which the pulse reaches its extreme value. This search is directed by the configured `polarity` and seeks

- the maximum, if `polarity` is set to `ADQ_POLARITY_POSITIVE`; and
- the minimum, if `polarity` is set to `ADQ_POLARITY_NEGATIVE`.

For each identified pulse, the analysis module determines

- the absolute value of the `peak`, measured relative to the `baseline`; and
- the `peak_position`, measured relative to the first sample in the source record.

The `peak` value is calculated as

$$\texttt{peak} = \begin{cases} x_{max} - \texttt{baseline} & \text{if } \texttt{polarity} = \texttt{ADQ\_POLARITY\_POSITIVE} \\ \texttt{baseline} - x_{min} & \text{if } \texttt{polarity} = \texttt{ADQ\_POLARITY\_NEGATIVE} \end{cases} \tag{10}$$

where x_{max} and x_{min} are the maximum or minimum values within the pulse boundary.

The `peak_position` is measured in samples relative to the first sample in the source record. While no ADC data propagates in a pulse attribute record, the timing information of the source record is copied to its `header` (see Fig. 15). Thus, the position of the `peak` on the digitizer's timing grid can be expressed as

$$t_{\texttt{peak}} = \texttt{timestamp} + \texttt{record\_start} + \texttt{peak\_position} \cdot \texttt{sampling\_period}. \tag{11}$$

Refer to Section 9.3 for general information about the `timestamp` and `record_start` header fields. Fig. 16 presents how the attributes `peak` and `peak_position` are calculated for a pulse when the `polarity` is set to `ADQ_POLARITY_POSITIVE`.

Note

If the same `peak` value is repeated for several samples, the `peak_position` will point to the *first* one.

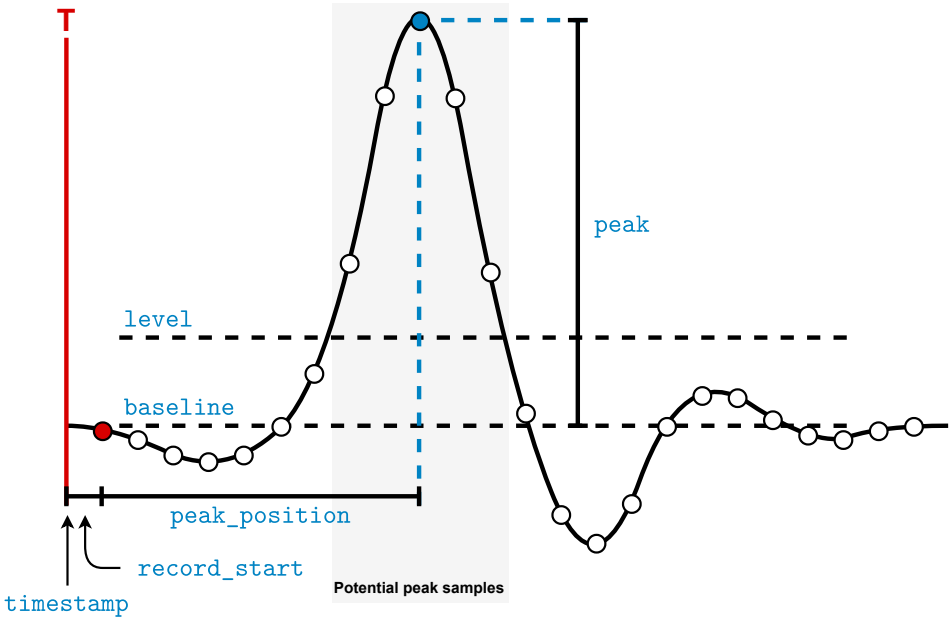


Figure 16: A detailed view of how the two attributes `peak` and `peak_position` are calculated for a pulse when the `polarity` is set to `ADQ_POLARITY_POSITIVE`. The `peak_position` is measured relative to the first sample in the record, whose position relative to the trigger event **T** is `record_start`.

Important

The `peak` value is given as a 16-bit *unsigned* integer to support the fact that a `dc_offset` can be applied to better utilize the available dynamic range. Peaks

- below the `baseline`, for `ADQ_POLARITY_POSITIVE`; and
- above the `baseline`, for `ADQ_POLARITY_NEGATIVE`

cannot be represented and indicates an incorrectly configured system.

5.7.6 Full Width at Half Maximum (FWHM)

The *full width at half maximum* (FWHM) is calculated by taking the horizontal difference between the two points at which the pulse is at half its amplitude. The amplitude is given by the `peak` value which is measured relative to the `baseline` and according to the `polarity`.

Since this calculation requires that the pulse is observed in its entirety to find the `peak`, its implementation relies on a memory. This memory has a finite size, which directly translates into a limitation for the maximum length of a pulse for which the `fwhm` can be correctly calculated. See Table 5 for details. The `fwhm` will be invalid for pulses exceeding this maximum length and the attributes will be marked as such in the `status` field.

The `fwhm` calculation does *not* interpolate at the intersections to increase the precision of the result. A threshold is set at half the `peak` value and the horizontal difference between

- the first sample at or after the threshold at the leading edge; and
- the first sample at or after the threshold at the trailing edge

is calculated to arrive at the `fwhm`. Thus, the attribute is measured in samples and given as a whole number of sampling periods. This can also be viewed as counting the number of samples between the two crossings of the threshold `peak` / 2. However, only samples above the `level` are considered, i.e. the samples *within* the pulse boundary. For that reason, the `level` should be set to less than half of the expected `peak` value in general.

Important

The FWHM calculation only considers samples *within* the pulse boundary. In other words, for a sample $x[n]$ to contribute to the `fwhm` value, it must fulfill

- $x[n] \geq \frac{\texttt{peak}}{2}$ *and* $x[n] \geq$ `level` for `ADQ_POLARITY_POSITIVE`; and
- $x[n] \leq \frac{\texttt{peak}}{2}$ *and* $x[n] \leq$ `level` for `ADQ_POLARITY_NEGATIVE`.

Fig. 17 demonstrates how the `fwhm` attribute is calculated when the `polarity` is set to `ADQ_POLARITY_POSITIVE`.

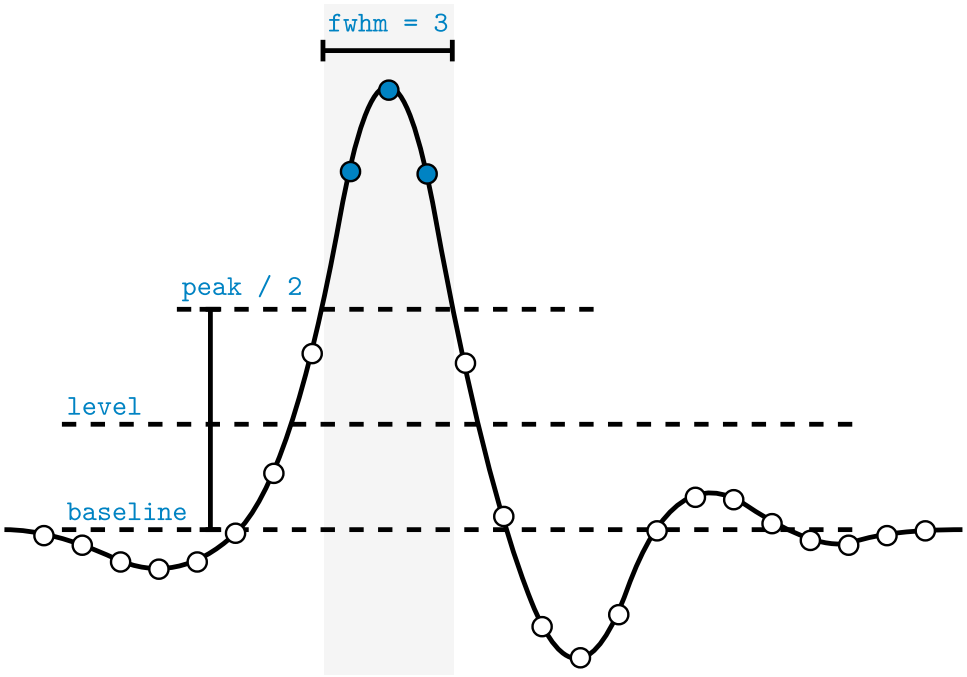


Figure 17: A detailed view of how the attribute `fwhm` is calculated for a pulse when the `polarity` is set to `ADQ_POLARITY_POSITIVE`.

5.7.7 Area

The *area* of a pulse is calculated from the samples within the pulse boundary with additional contribution from two adjacent static windows. The length of these windows can be configured independently from each other via the parameters `area_leading_edge_window_length` and `area_trailing_edge_window_length`. These windows may overlap with the ones extending from other pulses without consequences to the data flow. However, note that pulses in close proximity may contribute to each other's area.

Let the first sample in the leading edge window be $x[0]$ and the last sample in the trailing edge window be $x[N-1]$. The `area` is calculated as

$$\texttt{area} = \begin{cases} \sum_{n=0}^{N-1} (x[n] - \texttt{baseline}) & \text{if } \texttt{polarity} = \texttt{ADQ\_POLARITY\_POSITIVE} \\ \sum_{n=0}^{N-1} (\texttt{baseline} - x[n]) & \text{if } \texttt{polarity} = \texttt{ADQ\_POLARITY\_NEGATIVE} \end{cases} \tag{12}$$

meaning that it is always *positive* in the direction determined by the `polarity`. However, negative values are still possible for short pulses where the edge windows contain samples below the `baseline` or if the `baseline` is incorrectly configured. Fig. 18 demonstrates how the `area` of a pulse is calculated when the `polarity` is set to `ADQ_POLARITY_POSITIVE`.

If the length of the pulse exceeds the maximum limit (Table 5), the `area` calculation is invalid and will be marked as such in the `status` field.

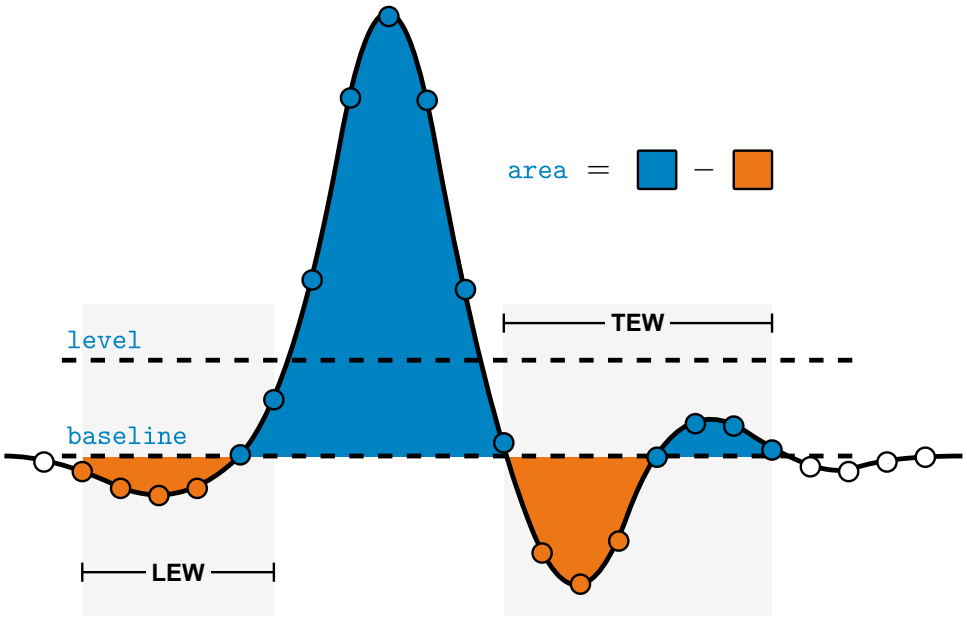


Figure 18: A detailed view of how the `area` is calculated for a pulse when the `polarity` is set to `ADQ_POLARITY_POSITIVE`. The samples contribute to the area relative to the `baseline`, see (12). In addition to the samples within the pulse boundary, the leading edge window (LEW) and the trailing edge window (TEW) determine which samples to include in the calculation.

5.7.8 Examples

This section introduces two configurations available as software examples specific to the FWPD firmware. These demonstrate how to combine the features of the digitizer for effective system-level solutions for pulse based applications. Both configurations focus on pulse analysis, but differ in how the records frame the ADC data, i.e. how the data acquisition process is configured. An assumption made in both cases is that the wider system features a well-defined, periodic signal that acts as a scheduler and indicates when useful pulse data is expected. This signal will be used to define the detection window.

- The first configuration demonstrates a case where the entire detection window is spanned by a single record with *static* length.
- The second configuration demonstrates a case where the detection window is implemented using the trigger blocking mechanism (Section 9.5) and each pulse generates a record with *dynamic* length.

Note

Both configurations are implemented in the software example `fwpd`. This example code is available in both C++ and Python in the release archive. See Section 15.2 for details on how to locate the example code.

Static Record Length

This case is presented in Fig. 15 (page 41) and relies on the following base configuration for the data acquisition process:

- The `trigger_source` for each channel is set to the event source matching the port where the external scheduler signal is connected to the digitizer, e.g. `ADQ_EVENT_SOURCE_TRIG`.
- The `trigger_edge` is set to the correct edge sensitivity for the scheduler signal.
- Dynamic record length is disabled: `dynamic_record_length_enabled` set to zero (default value).
- The `record_length` is set to the desired length of the detection window.
- The trigger blocking mechanism is disabled: `trigger_blocking_source` is set to `ADQ_FUNCTION_INVALID` (default value).

The records generated by the source channel and its associated attribute channel are matched one for one and the source record coincides *exactly* with the detection window. As a result, pulses within the detection window automatically have a common frame of reference in regards to how the `peak_position` is measured.

However, one drawback is that transferring data from the source channels will result in the *entire* detection window being transferred, since that is how a record is defined. This can potentially result in a high data rate since a large part of the bandwidth is spent transferring the "silent" parts of the detection window.

Thus, this method is *only* superior when *only* transferring pulse attribute data, as this negates the drawback by not transferring ADC data at all. Alternatively, the situation can be improved—at the cost of some configuration complexity—by instead starting from a base configuration implementing zero suppression, as described in the following section.

Dynamic Record Length

This case builds upon the zero suppression implementation discussed in Section 9.1.3. The acquisition configuration presented in that section results in each pulse being acquired as its own record, except for bursts, which may extend the record to include more than one pulse, see Fig. 42 (page 99).

Note

Refer to Section 9.1.3 for an overview of zero suppression for pulse data using FWDAQ.

The example configuration presented in this section utilizes

- the trigger blocking mechanism (Section 9.5) to implement a detection window on top of this acquisition pattern,
- zero length records (Section 9.5.1) to signal the absence of pulses within a detection window; and
- the timestamp synchronization mechanism (Section 7.3) to use the scheduler signal as a reference point for the `timestamp` of each record.

The general behavior of this configuration is presented in Fig. 19. Compared to the static length case, this configuration improves the transfer efficiency for the source channels, at the cost of slightly worse framing of the attribute data belonging to a single detection window (from the user's point of view). This attribute data will now be given over the course of several records and the `timestamp_synchronization_counter` will have to be queried to keep track of the grouping. Equation (11) can still be used to accurately position a `peak` on the digitizer's timing grid, but relies on the continuous synchronization of the digitizer's time base to the scheduler signal. The base configuration is as follows:

- The `trigger_source` for each channel is set to its own signal level event source, i.e. `ADQ_EVENT_SOURCE_LEVEL`.
- The `trigger_edge` is set to the edge sensitivity that matches the leading edge of a pulse:
  - `ADQ_EDGE_RISING` for positive pulses; and
  - `ADQ_EDGE_FALLING` for negative pulses.

- Dynamic record length is enabled: `dynamic_record_length_enabled` set to 1.
  - The lengths of the two edge windows: `dynamic_leading_edge_window_length` and `dynamic_trailing_edge_window_length` are set to appropriate values for the expected pulse shapes (refer to Section 9.1.1 for details).
  - The maximum length of a record: `dynamic_record_length_max` is set to a sensible value (refer to Section 9.1.2 for details).
- Zero length records (Section 9.5.1) are enabled with `zero_length_records_enabled` set to 1. If a detection window ends without having observed a pulse, a record with *no data* is emitted to signal this event.
- The timestamp synchronization mechanism is enabled with
  - the `source` set to the event source matching the port where the external scheduler signal is connected to the digitizer, e.g. `ADQ_EVENT_SOURCE_TRIG`,
  - the `edge` set to the correct edge sensitivity for the scheduler signal,
  - the `mode` set to `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_ALL`; and
  - the `arm` behavior set to `ADQ_ARM_AT_ACQUISITION_START`.
- The trigger blocking mechanism is enabled and `trigger_blocking_source` is set to one of the pattern generators, e.g. `ADQ_FUNCTION_PATTERN_GENERATOR0`.
- The pattern generator used as the `trigger_blocking_source` is configured to output
  - logic high until detecting an event of the scheduler signal with the correct edge sensitivity; and

```
instruction[0].source = ADQ_EVENT_SOURCE_TRIG;
instruction[0].source_edge = ADQ_EDGE_RISING;
instruction[0].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
instruction[0].output_value = 1;
instruction[0].output_value_transition = 0;
```

  - logic low for the duration of a detection window.

```
instruction[1].op = ADQ_PATTERN_GENERATOR_OPERATION_TIMER;
instruction[1].output_value = 0;
instruction[1].output_value_transition = 1;
```

Refer to Section 7.1 for general information about the pattern generators.

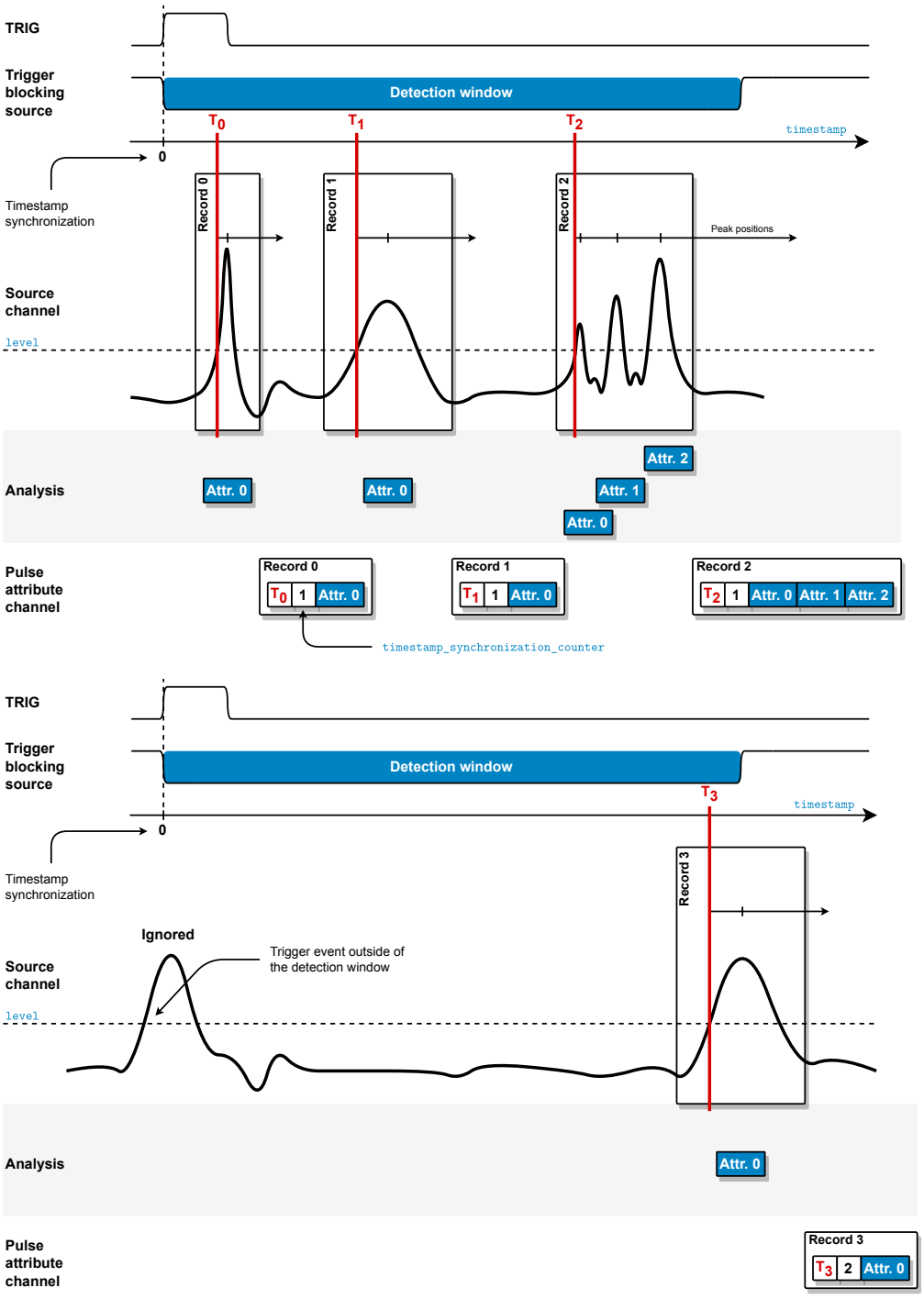


Figure 19: A system scheduler signal is connected to the TRIG port and used to define a detection window via the trigger blocking mechanism. The digitizer's time base is synchronized on the same event to create a common reference point for the `timestamp` of each record in the detection window. The `timestamp_synchronization_counter` reported in the `header` for each record increments for each detection window, allowing straightforward grouping of record data in the user application. A pulse whose trigger event fall outside the detection window is ignored. A pulse which begins inside the window but ends outside of it is analyzed in its entirety. This differs from the static length case in Fig. 15 (page 41).

6 Event Sources

Events symbolize the *availability of information* and may be used by various parts of the digitizer to accomplish certain tasks. They are generated by *event sources* and play a central role in defining how the digitizer acquires data, and how the supporting functions, e.g. timestamp synchronization (Section 7.3), behave during data acquisition and in general.

Event sources are identified using values from the enumeration `ADQEventSource`. Not all event sources in the enumeration are supported by an ADQ3 series digitizer. The available event sources are described in the following sections.

6.1 Trigger Events

The term *trigger event* (also: *trigger*) is reserved to specifically mean the event that triggers an acquisition of data. For example, the timestamp synchronization mechanism is stimulated by an event source, not a trigger source. However, the same event source may be selected to generate trigger events for channel A. In that context, it is a trigger source.

Not every event source may be selected as a trigger source. It is possible for an event source to exist solely to support a specific function and not relate to the data acquisition process directly. Fig. 20 illustrates this relationship.

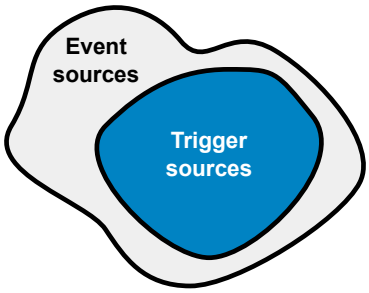


Figure 20: A trigger event always originates from an event source, but an event source may not always be used to generate trigger events.

6.2 Software

The software controlled event source allows the user to generate events from the user application. Call `SWTrig()` to generate an event. This event source is identified by the value `ADQ_EVENT_SOURCE_SOFTWARE` and may be used to generate trigger events.

Important

This event source only generates events with the rising edge polarity. Any consumer of these events must use the edge sensitivity `ADQ_EDGE_RISING`.

Important

There is *no* guarantee on the timing of events generated by the software controlled event source. Software events issued back-to-back may experience a large variation in their relative timing. This is in large part due to the scheduling performed by the operating system. The digitizer and the host computer does not constitute a real-time system.

6.3 Periodic

This event source generates events from the rising and falling edges of a clock signal that is synchronized to the sampling clock. There are three different methods of configuring the properties of the clock signal, specifying either:

- the logic `high` and logic `low` durations,
- the `period`; or
- the `frequency`.

The two latter methods yield a clock signal with approximately 50% duty cycle. This event source may be used to generate trigger events but is useful in other contexts as well. For example, the pulse generators (Section 7.2) can be stimulated by these events to synthesize a periodic digital signal that may be output on supported ports. The periodic event source is identified by the value `ADQ_EVENT_SOURCE_PERIODIC`.

Note

The resolution of the periodic event generator is equal to the digitizer's `sampling_frequency`, and is *unaffected* by any signal processing steps altering the effective sampling rate, e.g. sample skip (Section 5.2).

6.4 Signal Level

A signal level event source analyzes the post-processed data of an input channel, searching for points where the data crosses a target level.

Important

The dedicated signal level event sources for each channel can only be used by the data acquisition process to trigger records. The signal level event source matrix (Section 6.5) may additionally also be used by the pulse generators (Section 7.2).

Important

The signal level event source is only active during data acquisition, see Section 9.4.

The detection mechanism has two parameters (configurable per channel):

- the *signal level* (`level`); and
- the *arm hysteresis* (`arm_hysteresis`).

The purpose of the arming mechanism is to implement safeguard against incorrectly identifying events for slow-moving noisy signals. The arm hysteresis is a positive value indicating when the event detection should be armed and ready to identify either a rising or falling event. The hysteresis value is converted into an arm level for the respective event type as

$$\begin{aligned} \text{Arm level (rising)} &= \texttt{level} - \texttt{arm\_hysteresis} \\ \text{Arm level (falling)} &= \texttt{level} + \texttt{arm\_hysteresis} \end{aligned} \tag{13}$$

where the mechanism is armed to detect an event of each type at the first sample

- at or below the arm level for rising events; and
- at or above the arm level for falling events.

Fig. 21 presents a zoomed view of the rising edge of a slow-moving noisy signal. In this example, the arm hysteresis has been set too low, causing the event source to incorrectly output a second rising event as well as a falling event a short while later. Increasing the arm hysteresis will eliminate the false events. Thus, the arm hysteresis should be set to a value slightly higher than the signal's noise level.

Important

The arm hysteresis should be set to a value slightly higher than the signal's noise level.

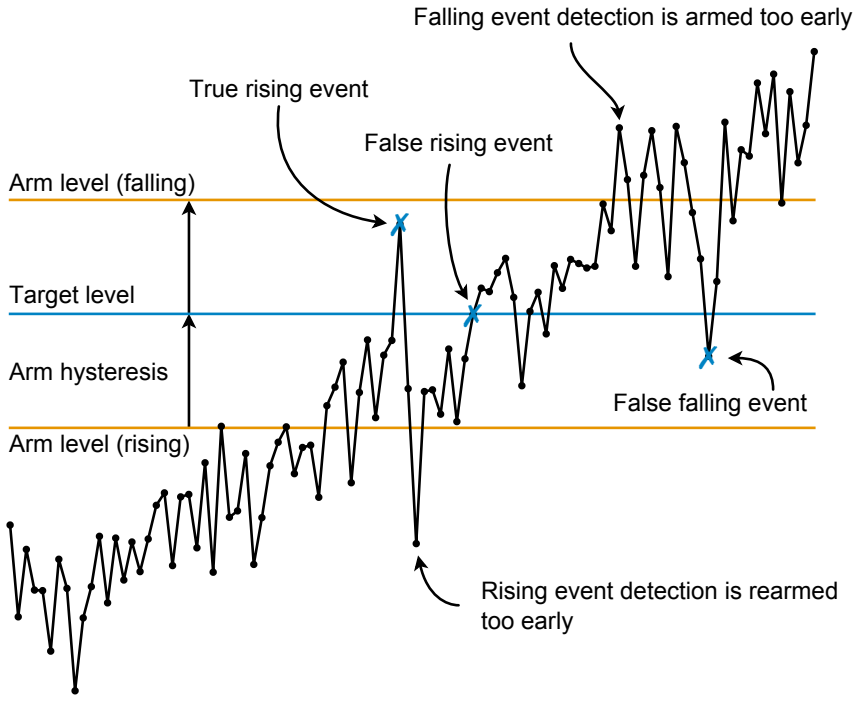


Figure 21: A demonstration of how a low arm hysteresis may cause false signal level events when analyzing a slow-moving noisy signal.

The signal level event source analyzing data from the first channel is identified by the value `ADQ_EVENT_SOURCE_LEVEL_CHANNEL0`, the second by `ADQ_EVENT_SOURCE_LEVEL_CHANNEL1` and so on. The

special value `ADQ_EVENT_SOURCE_LEVEL` exists as an alias for the channel itself and is only applicable in contexts where its use is not ambiguous. For example, the channel-specific data acquisition parameter `trigger_source` may be set to this value.

6.5 Signal Level Matrix

The signal level event source matrix allows the user to create a new event stream based on the event streams from any of the channel's dedicated signal level event sources. For each `channel` the user can specify whether or not the channel's signal level event stream is `enabled` and its `edge` sensitivity. Once configured, each event fulfilling the specification will be routed to the output. In some sense, the matrix computes logic OR between the input events.

This event source is identified by the value `ADQ_EVENT_SOURCE_LEVEL_MATRIX` and can be used by the acquisition process to trigger records, as well as by the pulse generators (Section 7.2) to construct output signals.

Important

The signal level event matrix can only be used as a trigger source and as input to the pulse generators (Section 7.2). It is important to note that there is a fixed nonzero latency between detecting the signal level event and changing the output from a pulse generator. While this latency is constant for repeated runs of the same digitizer configuration, the value may change if the digitizer firmware is updated.

Important

The signal level event source matrix is not to be confused with the matrix event source (Section 6.10). The two event sources allow a different set of inputs and cannot be combined.

Example

Consider a digitizer with two channels. Configure the signal level event source matrix to combine the rising edge events and falling edge events from both channels. Trigger channel A on rising edge events and channel B on falling edge events from the combined event stream:

```
struct ADQParameters adq;

/* Configure the signal level threshold for each channel. */
adq.event_source.level.channel[0].level = 500;
adq.event_source.level.channel[1].level = -2000;

/* Enable the event streams from channel A and channel B. Combine both types of
   edge events from both channels. */
adq.event_source.level_matrix.channel[0].enabled = 1;
adq.event_source.level_matrix.channel[0].edge = ADQ_EDGE_BOTH;
adq.event_source.level_matrix.channel[1].enabled = 1;
adq.event_source.level_matrix.channel[1].edge = ADQ_EDGE_BOTH;

/* Configure channel A to only trigger on rising edge events from the matrix. */
adq.acquisition.channel[0].trigger_source = ADQ_EVENT_SOURCE_LEVEL_MATRIX;
adq.acquisition.channel[0].trigger_edge = ADQ_EDGE_RISING;

/* Configure channel B to only trigger on falling edge events from the matrix. */
adq.acquisition.channel[1].trigger_source = ADQ_EVENT_SOURCE_LEVEL_MATRIX;
adq.acquisition.channel[1].trigger_edge = ADQ_EDGE_FALLING;
```

6.6 Port TRIG

The event source connected to the TRIG port performs edge detection on the input signal with a configurable voltage `threshold`. The timing precision for these events is higher for the TRIG port compared to other ports, e.g. SYNC. Additionally, it is the only port with timing precision higher than the base sampling rate of the digitizer, reaching *subsample* accuracy. For exact details on the specifications of the TRIG port when used as an event source, refer to the product datasheet [1] [2] [3] [4]. Refer to Section 8 for additional details on the TRIG port from a hardware perspective.

Example

For an ADQ32-PCIe digitizer running at 2500 MSPS, the TRIG port is sampled at 20 GSPS, i.e. eight times higher than the base sampling rate.

The TRIG event source is identified by the value `ADQ_EVENT_SOURCE_TRIG` and may be used as a trigger source.

6.7 Port SYNC

The event source connected to the SYNC port performs edge detection on the input signal with a configurable voltage `threshold`. The timing precision of the SYNC port is lower than the TRIG port. For exact details on the specifications of the SYNC port when used as an event source, refer to the product datasheet [1] [2] [3] [4]. Refer to Section 8 for additional details on the SYNC port from a hardware perspective.

Example

For an ADQ32-PCIe digitizer running at 2500 MSPS, the SYNC port is sampled at 312.5 MSPS, i.e. eight times lower than the base sampling rate.

The SYNC event source is identified by the value `ADQ_EVENT_SOURCE_SYNC` and may be used as a trigger source.

6.8 Port GPIOx

An ADQ3 series digitizer may feature one or several GPIO ports: GPIOA, GPIOB and so on. Certain pins in these ports may have an event source associated with them. Refer to Section 8.1 for information about the capabilities of the ports of a specific digitizer model.

The event source connected to a pin in a GPIO port performs edge detection on the input signal with a fixed voltage threshold. For exact details on the specifications of the event sources for GPIO signals, refer to the product datasheet [1] [2] [3] [4]. Refer to Section 8 for additional details on the GPIO ports from a hardware perspective.

Example

For an ADQ32-PCIe digitizer running at 2500 MSPS, pin 0 in the GPIOA port is sampled at 312.5 MSPS, i.e. eight times lower than the base sampling rate.

The event source tied to a GPIO pin is identified by its corresponding value in the enumeration `ADQEventSource`. For example, `ADQ_EVENT_SOURCE_GPIOA0` represents the event source associated with pin 0 in the GPIOA port.

6.9 Port PXIe

Digitizers featuring the PXIe interface have three additional event sources, one for each of the PXIe signals: STARB, TRIG0 and TRIG1. Refer to Section 8.1 for information about the ports of a specific digitizer model.

The event sources connected to pins in the PXIe port performs edge detection on its respective input signal with a fixed voltage threshold.

Example

For an ADQ36-PXIe digitizer running at 2500 MSPS, the PXIe STARB signal is sampled at 312.5 MSPS, i.e. eight times lower than the base sampling rate.

The event sources associated with the PXIe pins are identified by the values

- `ADQ_EVENT_SOURCE_PXIE_STARB`,
- `ADQ_EVENT_SOURCE_PXIE_TRIG0`; and
- `ADQ_EVENT_SOURCE_PXIE_TRIG1`.

Any of these event sources may be used as a trigger source.

6.10 Matrix

The matrix event source combines the event streams from a set of target event sources into a new event stream, essentially constructed as logic OR of the inputs—if any of the input sources produces an event, so will the matrix event source.

Each matrix `input` specifies a target event `source` and an `edge` sensitivity. The special value `ADQ_EVENT_SOURCE_INVALID` is used to specify a disabled input.

The order in which input sources are specified is important because it determines their relative priority, used in resolving conflicts when two input events occur in close proximity to one another. The source connected to input 0 has the highest priority followed by input 1 and so on. There are `ADQ_MAX_NOF_MATRIX_INPUTS` inputs, however only a subset of the digitizer's event sources can target the matrix. These are the event sources associated with external ports and the software controlled event source.

The matrix event source is identified by the value `ADQ_EVENT_SOURCE_MATRIX` and may be used to trigger the acquisition process and by any of the functions listed in Section 7.

Important

The matrix event source is not to be confused with the signal level event source matrix (Section 6.5). The two event sources allow a different set of inputs and cannot be combined.

Example

Configure the matrix event source to combine both the rising edge events and the falling edge events from the two ports TRIG and SYNC. Trigger channel A on rising edge events and channel B on falling edge events from the combined event stream.

```
struct ADQParameters adq;

/* Configure matrix input 0 (highest priority) to target both rising and falling
   edge events from the TRIG port. */
adq.event_source.matrix.input[0].source = ADQ_EVENT_SOURCE_TRIG;
adq.event_source.matrix.input[0].edge = ADQ_EDGE_BOTH;

/* Configure matrix input 1 (second highest priority) to target both rising and
   falling edge events from the SYNC port. */
adq.event_source.matrix.input[1].source = ADQ_EVENT_SOURCE_SYNC;
adq.event_source.matrix.input[1].edge = ADQ_EDGE_BOTH;

/* Other inputs are left disabled by keeping the default values. */

/* Configure channel A to only trigger on rising edge events from the matrix. */
adq.acquisition.channel[0].trigger_source = ADQ_EVENT_SOURCE_LEVEL_MATRIX;
adq.acquisition.channel[0].trigger_edge = ADQ_EDGE_RISING;

/* Configure channel B to only trigger on falling edge events from the matrix. */
adq.acquisition.channel[1].trigger_source = ADQ_EVENT_SOURCE_LEVEL_MATRIX;
adq.acquisition.channel[1].trigger_edge = ADQ_EDGE_FALLING;
```

6.11 Reference Clock Synchronization

Event sources associated with *external* ports and the software controlled event source can be synchronized to the digitizer's reference clock (see Section 4.2). When an event is synchronized to the reference clock, it is delayed until the next rising edge of the reference clock occurs. If several port events are detected during a reference clock period, the last event will be selected for synchronization and the earlier events ignored. This is illustrated in Fig. 22. The `reference_clock_synchronization_edge` parameter can be used to select which event type that should be synchronized. Valid values are:

- `ADQ_EDGE_RISING`, to only synchronize rising edge events (discarding falling edge events),
- `ADQ_EDGE_FALLING`, to only synchronize falling edge events (discarding rising edge events); and
- `ADQ_EDGE_BOTH`, to synchronize both types of events.

Normally, the edge should be set to the same value as the edge of the intended event source consumer. For example, if a synchronized event source is used to trigger the acquisition process, the specified `trigger_edge` should match the `reference_clock_synchronization_edge`. Synchronizing both types

of events and only consuming one type is also valid. Additionally, synchronizing events to the reference clock is only permitted when the following conditions hold true:

- ADQ30
  - 1CH-FWDAQ, 1CH-FWATD: sampling_frequency $\geq 16 \cdot$ reference_frequency
- ADQ32, ADQ33
  - 2CH-FWDAQ, 2CH-FWATD: sampling_frequency $\geq 16 \cdot$ reference_frequency
  - 1CH-FWDAQ, 1CH-FWATD: sampling_frequency $\geq 32 \cdot$ reference_frequency
- ADQ36
  - 4CH-FWDAQ: sampling_frequency $\geq 16 \cdot$ reference_frequency
  - 2CH-FWDAQ: sampling_frequency $\geq 32 \cdot$ reference_frequency

Important

The reference clock synchronization requires that the sampling frequency is either 16 or 32 times the reference frequency depending on firmware.

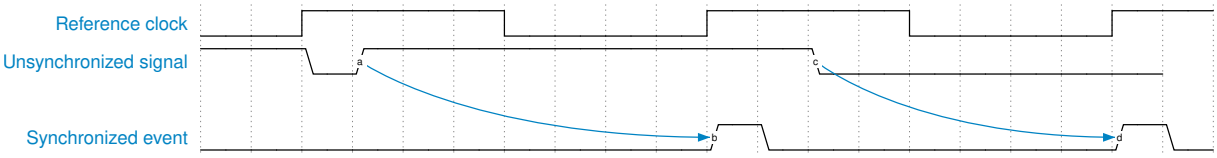


Figure 22: A timing diagram illustrating reference clock synchronization of an external signal. A rising edge event is captured and output for the first period, and a falling edge event is captured and output for the second. The edge specification is set to ADQ_EDGE_BOTH.

Important

The reference clock synchronization mechanism also features its own independent event source. It is identified by the value ADQ_EVENT_SOURCE_REFERENCE_CLOCK and only outputs rising edge events, marking the rising edge of the reference clock signal. This event source is always enabled.

7 Functions

A *function module* takes zero or more input signals and creates zero or more output signals. A function module with zero output signals implies that the effect is indirect, e.g. the timestamp synchronization module (Section 7.3) does not output any signal, instead it sets the digitizer's timestamp to a specific value. This definition is general by design to allow the feature set of the digitizer to grow organically over time, and to do so in a way that puts minimal strain on the mental model.

All function modules are *opt-in* and disabled by default. Configuring a function module is never required to acquire data. However, utilizing one or several may be required to achieve a desired system behavior, e.g. synchronizing the timestamp, blocking triggers or outputting a digital pulse pattern.

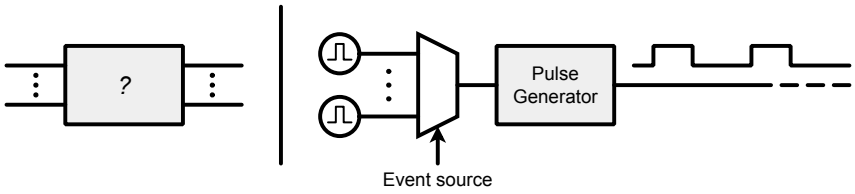


Figure 23: The definition of a function is general by design. They are opt-in and exist to meet specific use case requirements. One example is the pulse generator (Section 7.2) which is stimulated by an event source and outputs a digital pulse pattern. This signal can be output on any of the supported ports (Section 8).

7.1 Pattern Generator

The pattern generator module is capable of generating an arbitrary 1-bit pattern using counters and stimuli from the event source system (Section 6). The output signal may in turn be consumed by other parts of the digitizer to achieve a desired system behavior. A main function of the pattern generator is to use its output signal as stimuli for the trigger blocking mechanism (Section 9.5).

The pattern generator is configured with a set of instructions. For each instruction, the pattern generator can either output a logic high value (1) or a logic low value (0). The next instruction is loaded once the current instruction's transition condition is fulfilled: either a set time has passed, or a set number of events have been observed. The state of this transition may also be reset by an event. The number of instructions is limited to 16. While the pattern generator is active, the following loop is executed continuously:

1. Load the first instruction and output its logic value.
2. Wait until the transition condition is fulfilled. If the reset source is active and a reset event is detected, reload the current instruction and continue waiting.
3. If this is the last instruction, go to step 1. Otherwise, load the next instruction and go to step 2.

The pattern generator will automatically reset to step 1 when data acquisition is started with `StartDataAcquisition()`. This is done to facilitate the use of the pattern generator for trigger blocking (Section 9.5), which requires the pattern to be synchronized with the data acquisition process.

The pattern generator is enabled when `nof_instructions` is greater than zero. To disable the generator, set `nof_instructions` to zero. The digitizer contains several pattern generators which can be

used independently. The number of pattern generators available can be read from the parameter `nof_pattern_generators`. The parameters that constitute an instruction are described in the following sections.

7.1.1 Operation

The instruction's *operation* is specified with the parameter `op`. Each instruction can specify one of the following operations:

Event `ADQ_PATTERN_GENERATOR_OPERATION_EVENT`

An event instruction is active while waiting for a set number of events to be observed from the target `source`. The number of events to wait for is specified by the instruction parameter `count`.

Timer `ADQ_PATTERN_GENERATOR_OPERATION_TIMER`

A timer instruction is active for a set amount of time. The duration is specified by the instruction parameter `count`.

Note

An instruction can be made to never transition to the next by setting the operation to `ADQ_PATTERN_GENERATOR_OPERATION_EVENT` and the `source` to `ADQ_EVENT_SOURCE_INVALID`.

7.1.2 Count

The instruction parameter `count` determines the condition required to load the next instruction. The parameter serves different purposes depending on the operation. It either specifies

- the number of events to observe (event instruction); or
- a set amount of time measured in sampling periods (timer instruction).

For a timer instruction, the counter can be scaled using the `count_prescaling` parameter. The total count can be expressed as the product

$$\texttt{count\_prescaling} \cdot \texttt{count}. \tag{14}$$

Normally, the prescaling is used when the range of the `count` parameter is insufficient.

Example

On ADQ32, assuming a sampling rate of 2500 MSPS, the maximum count for a timer instruction corresponds to approximately 13 seconds with the prescaler set to 1 but can be extended up to 58 minutes with the prescaler set to its maximum value.

7.1.3 Source

The instruction parameters `source` and `source_edge` specify the event source and the edge sensitivity to observe for an event instruction. For timer instructions these parameters are ignored. Each instruction

source is independent of the other instructions but the total number of unique sources may not be larger than five per pattern generator.

Note

The maximum number of unique sources for each pattern generator is five.

7.1.4 Reset Source

The instruction parameters `reset_source` and `reset_source_edge` specify the event source and the edge sensitivity to observe for the reset condition. Each event observed from the reset source will reload the current instruction, effectively resetting the count and the transition state. For example, this can be used to *always* open a window with a fixed length on an event, regardless of whether a window is already open or not. See the corresponding example in Section 7.1.6. The reset source can be disabled by setting `reset_source` to `ADQ_EVENT_SOURCE_INVALID`. This is the default value.

7.1.5 Output Value

The logic value output by the pattern generator may change with each instruction. It is configured via the two parameters `output_value` and `output_value_transition`. The former defines the logic value to output during the whole instruction with the exception of the last cycle, where the logic value to output is defined by `output_value_transition`. This is illustrated in Fig. 24.

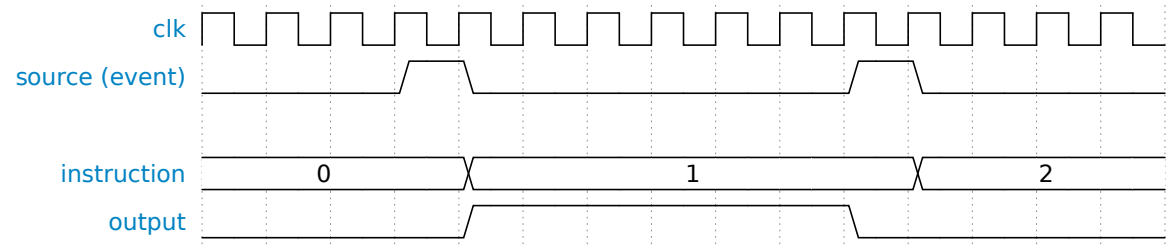


Figure 24: Timing diagram for two event instructions observing the same event source. Note the different transition values. Instruction 0 has `output_value` $= 0$, `output_value_transition` $= 0$ and instruction 1 has `output_value` $= 1$, `output_value_transition` $= 0$.

Important

While the `count` for a timer operation is specified in sampling periods, the granularity is not 1. This is because the pattern generator runs with a clock (`clk` in Fig. 24) that is slower than the sampling clock. Another consequence of this is that the position of events from event sources with resolution beyond this base clock will be truncated to the lower time resolution in the output signal.

7.1.6 Examples

This section provides a few examples of how to configure the pattern generator using the trigger blocking mechanism (Section 9.5) to provide a practical context. The source code snippets are written in the C programming language but the general concepts hold true regardless. The variable `parameters` is an `ADQPatternGeneratorParameters` struct and is assumed to be have been initialized with with a call to `InitializeParameters()`. Additionally, the examples assume that the pattern generator is selected as the trigger blocking source for a channel by setting the acquisition parameter `trigger_blocking_source` to target the configured pattern generator.

Example: once

Configure the pattern generator to output logic high (block all triggers) until a rising edge event is detected on the GPIOA port and then output logic low (accept all subsequent triggers). This behavior requires two instructions where the first is constructed as

```
parameters.nof_instructions = 2;
/* First instruction */
parameters.instruction[0].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
parameters.instruction[0].count = 1;
parameters.instruction[0].output_value = 1;
parameters.instruction[0].output_value_transition = 1;
parameters.instruction[0].source = ADQ_EVENT_SOURCE_GPIOA0;
parameters.instruction[0].source_edge = ADQ_EDGE_RISING;
```

The second instruction is set up to never transition since the use case requires that the output stays at a logic low level for the duration of the acquisition. This is accomplished by setting the `source` of an event instruction to `ADQ_EVENT_SOURCE_INVALID`.

```
/* Second instruction  */
parameters.instruction[1].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
parameters.instruction[1].count = 1;
parameters.instruction[1].output_value = 0;
parameters.instruction[1].output_value_transition = 0;
/* Set ADQ_EVENT_SOURCE_INVALID to ensure that the instruction never
   transitions */
parameters.instruction[1].source = ADQ_EVENT_SOURCE_INVALID;
```

Example: trigger count

Configure the pattern generator to output logic high (block all triggers) until a rising edge event has been observed on the SYNC port. Following this, output logic low (accept all triggers) until 100 rising edge events on the TRIG port have been observed. If a rising edge SYNC event occurs during the second phase, reset the counter and extend the acceptance window by an additional 100 rising TRIG events. This behavior requires two instructions where the first is constructed as

```
parameters.nof_instructions = 2;
/* First instruction */
parameters.instruction[0].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
parameters.instruction[0].count = 1;
parameters.instruction[0].output_value = 1;
parameters.instruction[0].output_value_transition = 1;
parameters.instruction[0].source = ADQ_EVENT_SOURCE_SYNC;
parameters.instruction[0].source_edge = ADQ_EDGE_RISING;
```

and the second instruction as

```
/* Second instruction */
parameters.instruction[1].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
parameters.instruction[1].count = 100;
parameters.instruction[1].output_value = 0;
parameters.instruction[1].output_value_transition = 0;
parameters.instruction[1].source = ADQ_EVENT_SOURCE_TRIG;
parameters.instruction[1].source_edge = ADQ_EDGE_RISING;
parameters.instruction[1].reset_source = ADQ_EVENT_SOURCE_SYNC;
parameters.instruction[1].reset_source_edge = ADQ_EDGE_RISING;
```

Example: window

Configure the pattern generator to output logic high (block all triggers) until a rising edge event has been observed on the TRIG port. Following this, output logic low (accept all triggers) during a window of 100000 sampling periods. Any trigger event occurring as the window is opened will be accepted due to the logic low transition value of the first instruction. This behavior requires two instructions where the first is constructed as

```
parameters.nof_instructions = 2;
/* First instruction */
parameters.instruction[0].op = ADQ_PATTERN_GENERATOR_OPERATION_EVENT;
parameters.instruction[0].count = 1;
parameters.instruction[0].output_value = 1;
parameters.instruction[0].output_value_transition = 0;
parameters.instruction[0].source = ADQ_EVENT_SOURCE_TRIG;
parameters.instruction[0].source_edge = ADQ_EDGE_RISING;
parameters.instruction[0].reset_source = ADQ_EVENT_SOURCE_INVALID;
```

The second instruction will allow rising TRIG events to restart the window, effectively extending the logic low output duration by another 100000 sampling periods.

```
/* Second instruction */
parameters.instruction[1].op = ADQ_PATTERN_GENERATOR_OPERATION_TIMER;
parameters.instruction[1].count = 100000;
parameters.instruction[1].output_value = 0;
parameters.instruction[1].output_value_transition = 0;
parameters.instruction[1].reset_source = ADQ_EVENT_SOURCE_TRIG;
parameters.instruction[1].reset_source_edge = ADQ_EDGE_RISING;
```

7.2 Pulse Generator

The pulse generator generates digital pulses with a fixed length on every event from the selected input event source. For the duration of the pulse, a logic high value is output, otherwise a logic low value is output. When an event from the selected event `source` matches the specified `edge` sensitivity, a pulse with a duration of `length` sampling periods is generated. Fig. 25 presents a conceptual block diagram.

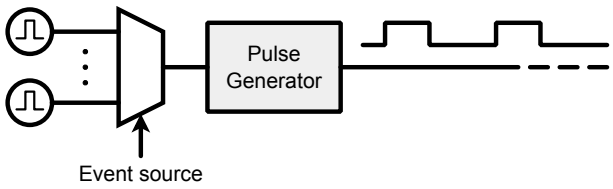


Figure 25: A conceptual block diagram of a pulse generator and its surrounding infrastructure.

The pulse generator signal can be output on one or several of the supported ports (Section 8). The output signal can be inverted on a port-basis via the port parameter `invert_output`.

The pulse generator may also be configured to *follow* the event source by setting the `length` parameter to -1. This effectively sets the output to logic high between the rising and falling events of the event source. There may be more than one pulse generator available to the user. These can be individually configured. The number of pulse generators available can be read programmatically from the parameter `nof_pulse_generators`.

Important

While the pulse `length` is specified in sampling periods, the granularity is not 1. This is because the pulse generator runs with a clock that is slower than the sampling clock. As a consequence, the position of events from event sources with resolution beyond this base clock will be truncated to the lower time resolution in the output signal.

Example

Consider an ADQ32 running at the base sampling frequency of 5 GHz (`length` granularity 32). Configure the periodic event source with a `period` of 800 samples, yielding an event frequency of 6.25 MHz. In this configuration, it is not possible to output a signal with *exactly* 50% duty cycle since the falling edge event occurs after 400 samples, which cannot be set as the pulse `length` since it is not divisible by 32. The `length` may be set to 384 samples (48% duty cycle) or to 416 samples (52% duty cycle). However, the period between rising edges will always be *exactly* 800 samples.

Important

If the source is set to `ADQ_EVENT_SOURCE_LEVEL_MATRIX` (Section 6.5), there will be a fixed nonzero latency between detecting the signal level event and changing the pulse generator's output. While this latency is constant for repeated runs of the same digitizer configuration, the value may change if the digitizer firmware is updated.

7.3 Timestamp Synchronization

The timestamp synchronization module is used to set the digitizer's internal timestamp counter (see Section 9.3) to a predefined value on an event. This can be used to establish a common time base for multiple digitizers by relating the time base to some external event. The value which the timestamp counter will be set to is configured with the `seed` parameter. There are three different modes, configured with the `mode` parameter:

Disable `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_DISABLE`

Timestamp synchronization is disabled. The timestamp counter is set to zero at power on.

First `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_FIRST`

Synchronize the timestamp on the first event. The timestamp counter is set to the `seed` value on the first event.

All `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_ALL`

Synchronize the timestamp on every event. The timestamp counter is set to the `seed` value for every event. A timing diagram for this mode is presented in Fig. 26.

The event is specified with the parameters `source` and `edge`. When an event is detected, the timestamp counter will be reset immediately *after* the event. This means that if a record is triggered on the same event, the record timestamp will have the value of the timestamp counter before the reset, not the `seed` value. The behavior is illustrated in the following example.

Example

Let the timestamp synchronization and the data acquisition trigger on the same periodic source with period T_p, and let the timestamp of the first record be T_0. The timestamp of a record n, T_n, is given by one of the following equations, depending on the timestamp synchronization `mode`:

$$T_{n,disable} = T_0 + n \cdot T_p \quad \forall n \tag{15}$$

$$T_{n,first} = \begin{cases} T_0 & \text{if } n = 0 \\ \texttt{seed} + n \cdot T_p & \text{if } n > 0 \end{cases} \tag{16}$$

$$T_{n,all} = \begin{cases} T_0 & \text{if } n = 0 \\ \texttt{seed} + T_p & \text{if } n > 0 \end{cases} \tag{17}$$

The timestamp synchronization can either be armed immediately when the parameters are written, or when the data acquisition is started. Note that in the latter case, there is still a *nonzero* time difference between the arming of the timestamp synchronization and the data acquisition process. The arm behavior is configured using the `arm` parameter.

The number of times the timestamp has been synchronized is available in the record header field `timestamp_synchronization_counter`. The value can also be read separately via a call to `GetStatus()`.

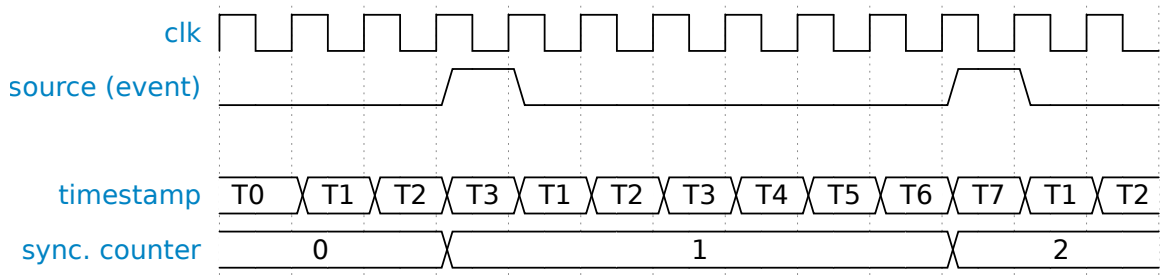


Figure 26: Timing diagram of the timestamp synchronization mechanism synchronizing on every event.

7.4 Daisy Chain

One problem facing a system consisting of more than one digitizer, where synchronicity is critically important, is how to distribute the trigger signal. Consider such a system where the main goal is to use a single trigger source to trigger the data acquisition process for all the channels of all the digitizers at the same time. If the system only contains a few digitizers, one potentially successful strategy is to connect the signal to each digitizer by first going through a splitter, followed by cables of equal length. However, this approach does not scale well with the number of digitizers in the system. In addition to the large amount of cables required, the trigger signal is eventually attenuated beyond the sensitivity of the analog inputs, prompting the use of supporting electronics such as active fanout buffers.

To address this problem, ADQ3 series digitizers supports *daisy chain triggering*. This is a *system level solution* that combines several features of the digitizer to acquire data across multiple digitizers simultaneously with a single trigger event. The name comes from the digitizers being connected to each other in series in a way that allows an electrical signal—the *daisy chain signal*—to propagate between them. Simultaneous data acquisition is achieved by using a common reference clock and propagating the daisy chain signal through all digitizers according to a fixed scheme. Optionally, the timing performance can be further improved by utilizing the clock system's capability of individually adjusting each digitizer's reference clock, see Section 4.2 for additional details.

At its core, the daisy chain trigger mechanism offers a solution to large scale trigger distribution for synchronous acquisition, converting a badly scaling problem in the analog domain (fanning out the trigger signal) into a more manageable problem in the digital domain.

7.4.1 Structure

The daisy chain itself is constructed by connecting an output-capable port of one digitizer to an input-capable port of another, repeating this pattern until the target digitizers have all been connected as links in a chain. Refer to Section 8 for more information about ports. Technically, each digitizer is allowed to propagate the daisy chain signal to several others as long as the signal level is within acceptable limits[1] of the input ports. In other words, a tree-like topology is supported and the default one-to-one (series) connection pattern can be considered as a special case.

Within the daisy chain, a digitizer will always assume one of the following distinct types:

- the *primary* type (position 0); or
- the *secondary* type (position 1, 2, ...).

Important

The implementation only supports a *single* primary digitizer, i.e. the daisy chain must be unidirectional; a ring topology is *not* supported.

The primary digitizer is placed first in the daisy chain and is responsible for capturing the trigger event. In the most straightforward configuration, the primary digitizer (position 0) will synchronize the trigger event to the reference clock and output a pulse on its designated output port. This signal will propagate to the first secondary digitizer (position 1), which triggers its own internal data acquisition process on the

[1] Refer to the product datasheet [1] [2] [3] [4]for additional details.

next rising edge of the reference clock before propagating the signal onwards, again synchronized to the reference clock. This continues until the last digitizer has received the daisy chain signal. Fig. 28 shows an example of such a configuration using several ADQ32-PCIe digitizers spread across two host computers. The figure is explained further in Section 7.4.4. A timing diagram of the data acquisition process with this propagation scheme is presented in Fig. 27.

Since each digitizer triggers at an increasingly later point in time, the `horizontal_offset` of each digitizer must be set up with increasingly larger values to correctly capture data around the trigger point of the primary digitizer. Since negative horizontal offset is implemented using memory, there is an upper limit to the length (measured in number of positions) of the daisy chain.

Note

With each advancement of a digitizer's *position* in the daisy chain comes a requirement of a certain minimum negative `horizontal_offset`. This shifts the parameter's lower bound upwards, restricting the range of horizontal adjustment available to the user.

The position of a digitizer within the chain is incremented each time the daisy chain signal is *resynchronized to the reference clock*. This is typically carried out when the signal passes through each digitizer, but is technically under the user's control. This can be leveraged to create topologies that greatly reduce the effective length of the chain, and by extension, the memory requirements. Compare the two examples in Sections 7.4.4 and 7.4.5, whose topologies are presented in Figs. 29 and 31.

Example

Consider a daisy chain set up such that:

- *Digitizer A* is the primary digitizer and outputs the daisy chain signal to *Digitizer B*, *Digitizer C* and *Digitizer D*.
- *Digitizer D* outputs the daisy chain signal to *Digitizer E* and *Digitizer F*.
- *Digitizer F* outputs the daisy chain signal to *Digitizer G*.

Assuming each device resynchronizes the chain to the reference clock by setting `resynchronization_enabled` to 1, the position of the devices are as follows:

Trigger source

- **Digitizer A** — *Position 0*
  - **Digitizer B** — *Position 1*
  - **Digitizer C** — *Position 1*
  - **Digitizer D** — *Position 1*
    - **Digitizer E** — *Position 2*
      - **Digitizer G** — *Position 3*
    - **Digitizer F** — *Position 2*

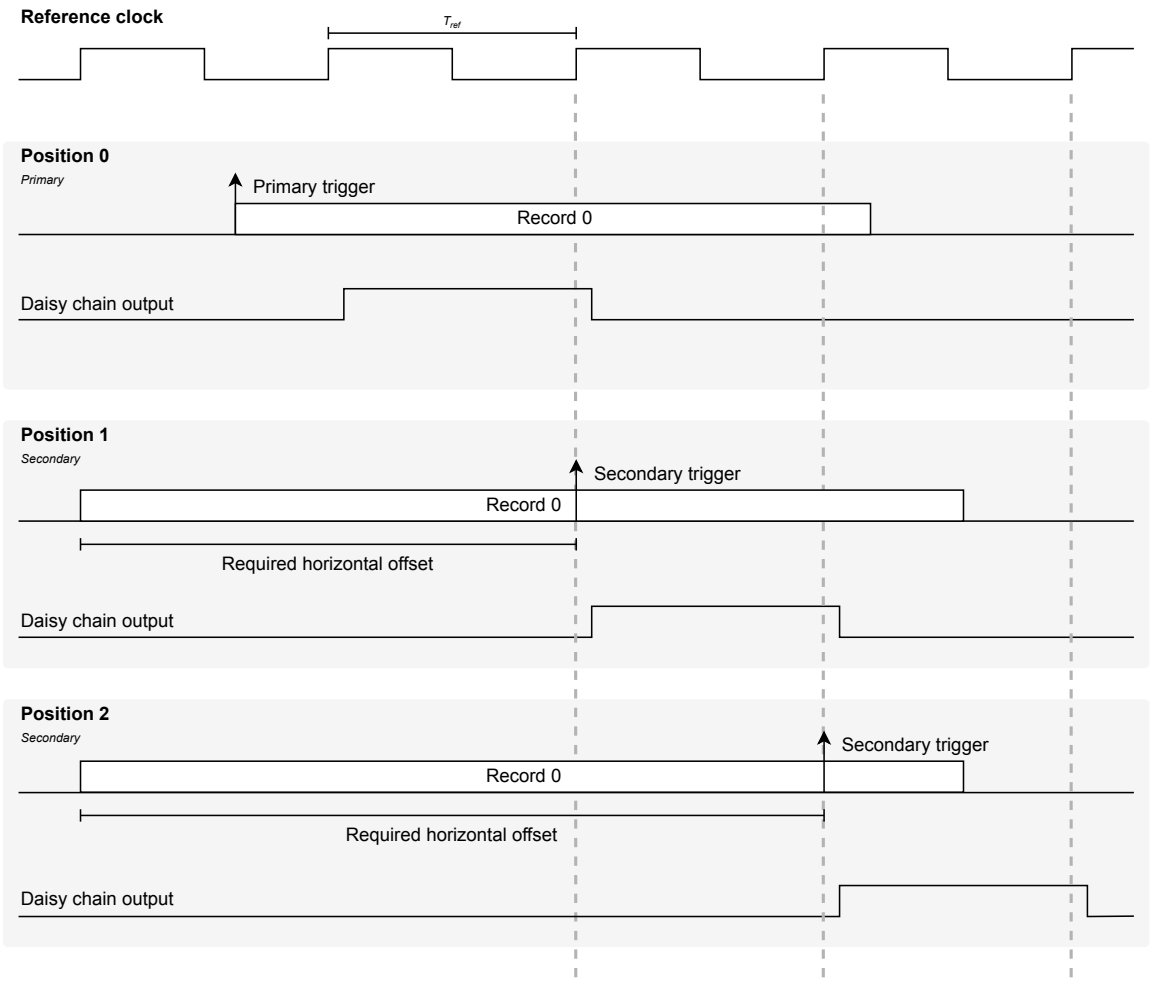


Figure 27: A high-level timing diagram of the data acquisition process for the daisy chain trigger mechanism in its most straightforward configuration. Refer to Section 7.4.1 for details.

7.4.2 Phase One: Synchronizing the Timing Grid

A fundamental requirement of the daisy chain mechanism is for all digitizers to share a common reference clock. While this makes it so the digitizers run at precisely the same rate, initially their timing grids are not synchronized, i.e. they do not agree on where $t = 0$. Thus, a prerequisite to entering the acquisition phase will be to synchronize the timing grids of all the digitizers in the system. This is accomplished by using

- the software controlled event source (Section 6.2),
- the seeding functionality of the timestamp synchronization module (Section 7.3); and
- the daisy chain and its precise propagation of a 1-bit digital signal.

The existing daisy chain will be used to propagate the synchronization signal, which will be triggered from the user application via the software controlled event source. The timestamp synchronization module for each digitizer will be seeded with a value corresponding to its position within the daisy chain. As

the signal propagates through the chain, the digitizers' timing grids sequentially restart from the seeded values which effectively *anticipate* the propagation delay, ultimately synchronizing the timing grids to agree on $t = 0$. Refer to steps 1–4 in Section 7.4.7 for details on how to perform this task.

When all the digitizers in the system agree on $t = 0$, relating events in time from any of them is a trivial operation. The `timestamp` of an acquired record will hold a value of *absolute time* and these values are precisely the same across the entire system.

Important

The synchronization of the timing grid must be performed each time the clock system is reconfigured (or initialized after power-up) for *any* of the digitizers in the system. However, as long as the clock system configuration remains fixed, this operation does not need to be repeated.

7.4.3 Phase Two: Continuous Operation

Once phase one (Section 7.4.2) has successfully been completed, the system is ready for continuous operation of the daisy chain. Apart from selecting the appropriate trigger source for the primary digitizer and switching the arming strategy, the configuration steps remain the same as phase one. Refer to steps 5–6 in Section 7.4.7 for configuration details.

The synchronized timing grids established in phase one trivializes the task of aligning records from any digitizer in the system. Since the semantics of each record's `timestamp` and `record_start` is preserved, the values can be used as normal to relate records to each other in time. Refer to Section 9.3 for additional details on the timing information of a record.

The pseudo code below demonstrates how to extract the samples of the secondary digitizer, `s_data`, that corresponds to the samples of the primary digitizer, `p_data`. The `time_unit` of the digitizers are assumed to be equal.

```
# The timestamp of the first sample of the primary device
p_ts = primary_record.header.timestamp + primary_record.header.record_start

# The timestamp of the first sample of the secondary device
s_ts = secondary_record.header.timestamp + secondary_record.header.record_start

# Get the index of the first sample after the trigger event for the secondary device
index = ceil((s_ts - p_ts)/secondary_record.header.sampling_period))

p_data = primary_record.data
s_data = secondary_record.data[index : index + record_length]
```

7.4.4 Example: ADQ32-PCIe

Consider a system of two host computers, each with four ADQ32-PCIe digitizers. Fig. 28 shows one example of how the system can be connected to utilize the daisy chain trigger mechanism, with the resulting topology presented in Fig. 29.

In line with the prerequisites, a common reference clock is distributed to each digitizer's CLK port (Section 8.5) through cables of *equal length*. The trigger signal is connected to the TRIG port of D0, which assumes the role of primary digitizer. The daisy chain signal propagates sequentially through the remaining (secondary) digitizers, entering on the TRIG port and exiting on the SYNC port. Each digitizer synchronizes the outbound signal to the reference clock, thus advancing the *position* of the next device by one. Refer to Fig. 27 for a timing diagram of how the daisy chain signal would propagate for the first three digitizers in this example.

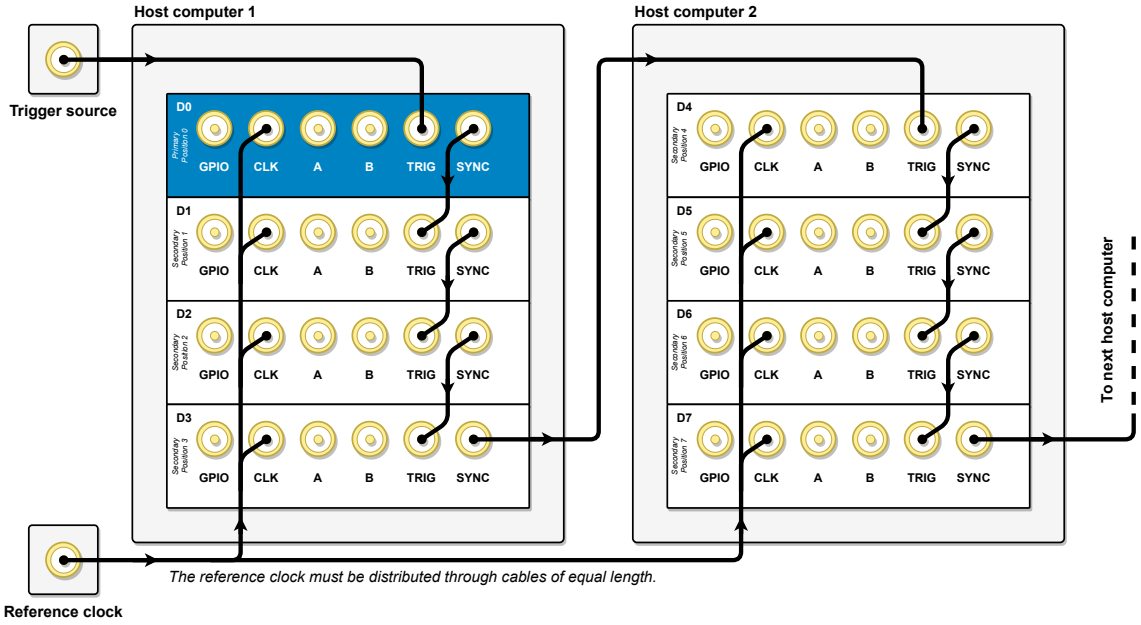


Figure 28: A block diagram of a system using the daisy chain mechanism to trigger eight ADQ32-PCIe digitizers (16 channels in total) across two PCIe-capable host computers. The trigger signal is connected to the TRIG port of D0, which assumes the role of primary digitizer. The daisy chain signal propagates sequentially through the remaining (secondary) digitizers, entering on the TRIG port and exiting on the SYNC port. Each digitizer synchronizes the outbound signal to the reference clock, thus advancing the *position* of the next device by one. It is crucial that the reference clock is distributed to each host computer through cables of equal length.

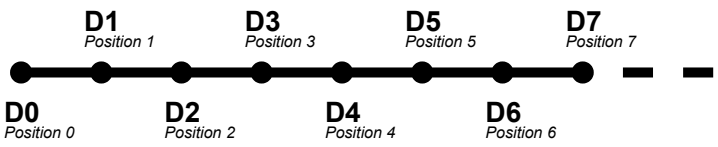


Figure 29: A diagram showing the resulting daisy chain topology from the configuration in Fig. 28.

7.4.5 Example: ADQ36-PXIe

One system layout that benefits significantly from the daisy chain trigger mechanism is multiple PXIe-chassis filled with ADQ36-PXIe. In such a system, both the reference clock and the daisy chain signal may be distributed through the PXIe backplane (within a chassis). An example configuration is presented in Fig. 30, with the resulting topology presented in Fig. 31.

Following the established prerequisites, a common reference clock is distributed to both chassis through cables of *equal length*. The trigger signal is connected to the TRIG port of D0, which assumes the role of primary digitizer. However, in this example, the trigger bus in the PXIe backplane is utilized to its full potential, neatly managing the propagation of the trigger signal between digitizers in a chassis without the need for cables between ports in the front panels. The daisy chain signal is passed from one chassis to the next via the SYNC ports of the outermost digitizers. In the second chassis, the receiving digitizer synchronizes the signal to the reference clock before the propagation through the PXIe backplane is repeated, this time moving from right to left.

Due to the topology afforded by propagating the daisy chain signal through the PXIe backplane (Fig. 31), multiple digitizers are located at the same position within the chain, which greatly reduces the memory requirements since the maximum `horizontal_offset` scales with the number of positions. Consider the timing diagram in Fig. 27:

- Digitizers D1–D7 will experience the signal propagation according to position 1, since they are effectively connected *in parallel*.
- Digitizer D8 will experience the signal propagation according to position 2.
- Digitizers D9–D15 will experience the signal propagation according to position 3, since they are effectively connected in parallel.

7.4.6 Limitations

The daisy chain mechanism has the following limitations:

- The *rearm time* is the shortest possible time between two trigger events on the primary digitizer for which the daisy chain functions correctly. This can also be expressed as a maximum trigger frequency, f_{max}, and is calculated as

$$f_{max} = \frac{\texttt{reference\_frequency}}{Q}, \tag{18}$$

  where Q is an integer calculated as

$$Q = \left\lceil \frac{\texttt{skip\_factor} \cdot \texttt{record\_length} \cdot \texttt{reference\_frequency}}{\texttt{sampling\_frequency}} + 1 \right\rceil. \tag{19}$$

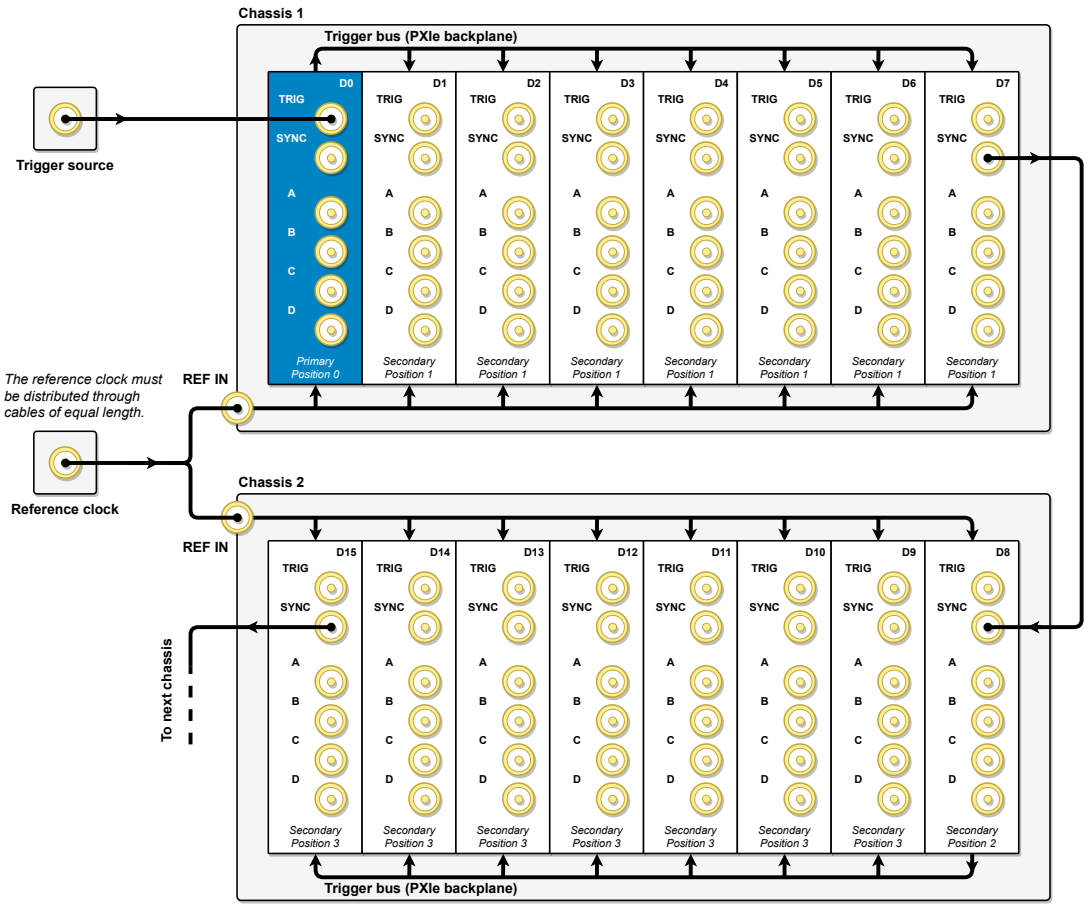


Figure 30: A block diagram of a system using the daisy chain mechanism to trigger 16 ADQ36-PXIe (64 channels in total) across two PXIe chassis. The trigger signal is input on the TRIG port of digitizer D0 and propagates through the chassis backplane before traveling between the chassis through the respective SYNC ports of digitizers D7 and D8. Digitizer D8 resynchronizes the signal to the reference clock before passing it through the chassis backplane, this time from right to left. It is crucial that the reference clock is distributed to each chassis through cables of equal length.

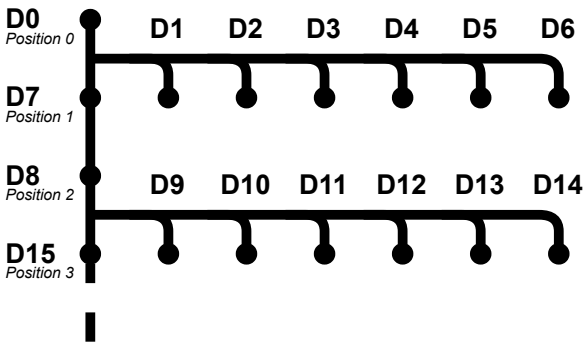


Figure 31: A diagram showing the resulting daisy chain topology from the configuration in Fig. 30.

Example

Consider an ADQ32-PCIe running its two-channel firmware at the base sampling rate with an external 10 MHz reference clock. The record length is set to 1024 samples and the sample skip factor to 1 (no samples are discarded). Using (18) and (19), the maximum trigger rate f_{max} can be determined as

$$Q = \left\lceil \frac{1 \cdot 1024 \cdot 10 \cdot 10^6}{2.5 \cdot 10^9} + 1 \right\rceil = \lceil 5.096 \rceil = 6$$

$$f_{max} = \frac{10 \cdot 10^6}{6} \approx 1.67 \text{ MHz}$$

The rearm time is

$$T_{rearm} = \frac{1}{f_{max}} = \frac{6}{10 \cdot 10^6} = 600 \text{ ns}$$

- The clock system configuration must fulfill the following criteria to guarantee correct operation of the daisy chain trigger mechanism:

$$T_{propagation} + T(\texttt{sampling\_frequency}) < T_{ref}, \tag{20}$$

where

  - $T_{propagation}$ is a constant that specifies the time it takes to propagate the daisy chain signal between two neighboring digitizers. This value is system dependent since it includes propagation through cables and analog signal buffers, but normally lies in the nanosecond range.
  - $T(\texttt{sampling\_frequency})$ is a function of the nominal sampling frequency that also depends on the digitizer and its firmware. See Table 6.
  - T_{ref} is the reference clock period, i.e. $1/\texttt{reference\_frequency}$.

Table 6: $T(\texttt{sampling\_frequency})$ is a function of the nominal sampling frequency that varies with the digitizer model and its firmware.

Model	Firmware	$T(\texttt{sampling\_frequency})$
ADQ30	1CH	$15 \cdot 8$ / sampling_frequency
ADQ32	2CH	$15 \cdot 8$ / sampling_frequency
	1CH	$15 \cdot 16$ / sampling_frequency
ADQ33	2CH	$15 \cdot 8$ / sampling_frequency
ADQ36	4CH	$15 \cdot 8$ / sampling_frequency
	2CH	$15 \cdot 16$ / sampling_frequency

Example

Consider an ADQ36-PXIe running its two-channel firmware (base sampling rate 5 GHz). Assume a propagation delay of $\approx$ 5 ns and a desire to use an external reference clock input on the CLK port. Using (20) and Table 6, the minimum reference clock period is determined as

$$5 \cdot 10^{-9} + \frac{15 \cdot 16}{5 \cdot 10^9} < T_{ref} \quad \Rightarrow \quad T_{ref,min} = 5 \cdot 10^{-9} + \frac{15 \cdot 16}{5 \cdot 10^9} = 5.3 \text{ ns},$$

and thus the maximum reference clock frequency is

$$f_{ref,max} = \frac{1}{T_{ref,min}} \approx 18.9 \text{ MHz}.$$

Note that the clock system places additional constraints on the reference clock frequency so not every value in this range will be accepted.

Example

Consider an ADQ32-PCIe running its one-channel firmware (base sampling rate 5 GHz) with its internal 10 MHz reference clock. Assume a propagation delay of $\approx$ 5 ns. Using (20) and Table 6 gives

$$5 \cdot 10^{-9} + \frac{15 \cdot 16}{f_s} < \frac{1}{10 \cdot 10^6} \quad \Rightarrow \quad f_s > \frac{15 \cdot 16}{\left(\frac{1}{10 \cdot 10^6} - 5 \cdot 10^{-9}\right)},$$

and thus the minimum sampling frequency is

$$f_{s,min} = \frac{15 \cdot 16}{\left(\frac{1}{10 \cdot 10^6} - 5 \cdot 10^{-9}\right)} \approx 2.53 \text{ GHz}.$$

Note that the clock system places additional constraints on the sampling frequency so not every value in this range will be accepted.

- To keep the sampling points for all digitizers in the system well-aligned, the following expression must hold true:

$$\frac{\texttt{sampling\_frequency}}{\texttt{reference\_frequency}} \in \mathbb{N}, \tag{21}$$

  i.e. the reference frequency must evenly divide the sampling frequency in order for these two clock signals to be in phase with each other.

Important

- The `sampling_frequency` must be the same for all digitizers in the system.
- The `reference_frequency` must be the same for all digitizers in the system.

If the sample skip module (Section 5.2) is enabled and thus performs additional data rate reduction,

the expression in (21) is extended to

$$\frac{\texttt{sampling\_frequency}}{\texttt{reference\_frequency} \cdot \texttt{skip\_factor}} \in \mathbb{N}, \tag{22}$$

where the `skip_factor` is allowed to vary between digitizers in the system. This requirement is not a consequence of the daisy chain mechanism itself, but rather a reality of synchronizing a system with more than one digitizer running with different effective sampling rates.

Note

Technically, there is nothing preventing the user from specifying a `skip_factor` that violates the condition in (22) but is otherwise accepted by the digitizer. The daisy chain mechanism will continue to work correctly in these situations since the timing grids are aligned. However, the digitizers' sampling grids will not always align and more importantly, may experience different phases for each restarted acquisition. Regardless, the expression in (21) must never be violated.

7.4.7 Configuration

While the term *daisy chain* is primarily used to refer to the concept as a whole, in the context of configuring the digitizer, it also refers to the digitizer's dedicated function module. The parameters for this module are defined by `ADQDaisyChainParameters`. Since daisy chain trigger mechanism is a system level solution, just configuring a single function module will not successfully trigger all the digitizers in the chain. The list below outlines the required configuration steps and will have to be implemented by the user at the system level.

1. Configure the ports used for input and output of the daisy chain signal, see Section 8.
2. Configure the daisy chain for timestamp synchronization.
    - For the primary digitizer, set the daisy chain `source` to `ADQ_EVENT_SOURCE_SOFTWARE` and the `edge` to `ADQ_EDGE_RISING`.
    - For secondary digitizers, set the daisy chain `source` to the event source associated with the daisy chain input port.
    - For all digitizers, set the `position` in the daisy chain. This value starts at 0 for the primary digitizer and increments by one each time the daisy chain signal is resynchronized to the reference clock.

      Important

      Multiple digitizers may have the same `position` within the chain, depending on the topology.

    - For all digitizers, set the daisy chain `arm` parameter to `ADQ_ARM_IMMEDIATELY`.
    - For all digitizers, set the daisy chain `resynchronization_enabled` parameter to 1.
    - Enable reference clock synchronization for the daisy chain `source`, see Section 6.11.
        - For the primary digitizer, enable reference clock synchronization for the software controlled event source.

  - For secondary digitizers, enable reference clock synchronization for the event source associated with the digitizer's daisy chain input port.
- For all digitizers, set the timestamp synchronization `mode` to `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_FIRST`.
- For all digitizers, set the timestamp synchronization `arm` parameter to `ADQ_ARM_IMMEDIATELY`.
- Set the timestamp synchronization `seed` to

$$\texttt{seed} = 16 \cdot \texttt{position} \cdot T_{ref,samples} \tag{23}$$

where $T_{ref,samples}$ is the period of the reference clock in samples. This value can be calculated using the clock system parameters as

$$T_{ref,samples} = \frac{\texttt{sampling\_frequency}}{\texttt{reference\_frequency}}. \tag{24}$$

This is guaranteed to be an integer due to the prerequisite discussed in Section 7.4.6.

3. For the primary digitizer, issue the timestamp synchronization event by calling `SWTrig()`.
4. Read the timestamp synchronization status (pass `ADQ_STATUS_ID_TIMESTAMP_SYNCHRONIZATION` to `GetStatus()`) for each digitizer and verify that the `counter` has been incremented by one.
5. Configure the daisy chain for data acquisition.
   - For the primary digitizer, set the daisy chain `source` and `edge` to the desired trigger source. To trigger on a signal level, the signal level matrix (`ADQ_EVENT_SOURCE_LEVEL_MATRIX`) must be used, see Section 6.5. The regular signal level event source is not supported.
   - For secondary digitizers, set the daisy chain `source` to the port used for daisy chain input.
   - For all digitizers, set the `position` in the daisy chain. This value starts at zero for the primary digitizer and increments by one each time the daisy chain signal is resynchronized to the reference clock.

     Important

     Multiple digitizers may have the same `position` within the chain, depending on the topology.

   - For all digitizers, set the daisy chain `arm` parameter to `ADQ_ARM_AT_ACQUISITION_START`.
   - For secondary digitizers, enable reference clock synchronization for the daisy chain `source`, see Section 6.11. The reference clock synchronization should not be enabled for the primary digitizer unless the that behavior is specifically desired.
6. Configure the data acquisition process.
   - Set the `trigger_source` and `trigger_edge` to the values specified in step 5, i.e. the daisy chain module's `source` and `edge`.
   - Set the `horizontal_offset` to

$$\texttt{horizontal\_offset} = \begin{cases} \frac{T_{ref,samples}(1+\texttt{position})-L}{\texttt{skip\_factor}} & \texttt{position} > 0 \\ 0 & \texttt{position} = 0 \end{cases} \tag{25}$$

where $T_{ref,samples}$ is the period of the reference clock in samples (24), and L is an extra offset that is required if the signal level event source matrix is used. The values of L are listed in Table 7. For all other event sources, L should be set to zero. The `skip_factor` is the sample skip factor, see Section 5.2.

Important

The `horizontal_offset` must be set to nearest supported value greater than or equal to the computed value.

- Set the record length to

$$\texttt{record\_length} = \begin{cases} N + T_{ref,samples}, & \texttt{position} > 0 \\ N & \texttt{position} = 0 \end{cases} \tag{26}$$

where N is the desired record length in samples, $T_{ref,samples}$ is the period of the reference clock in samples (24). This addition ensures that the data captured by the secondary digitizers overlap the data capture by the primary digitizer (see Fig. 27).

Table 7: Additional horizontal offset required for secondary digitizers in the daisy chain when the signal level event source matrix (`ADQ_EVENT_SOURCE_LEVEL_MATRIX`) is used.

Model	Firmware	Extra horizontal offset L
ADQ30	1CH	680 samples
ADQ32	2CH	680 samples
	1CH	1360 samples
ADQ33	2CH	680 samples
ADQ36	4CH	680 samples
	2CH	1360 samples

7.4.8 Runtime Error Reporting

During operation, a digitizer continuously monitors its own segment of the daisy chain for two main issues:

- a `setup_time_warning`, indicating that a secondary digitizer has received an edge of the daisy chain signal close to the edge of the reference clock; and
- a `rearm_error`, indicating that the trigger source selected for the primary digitizer emitted a new trigger event during the rearm period.

These two status conditions are accessed individually for each digitizer through the status reporting mechanism `GetStatus()` by passing the identifier `ADQ_STATUS_ID_DAISY_CHAIN`. They are both binary conditions and *sticky*, indicating that an issue has occurred at some point earlier in time. The values are cleared after each call to `GetStatus()`.

The `setup_time_warning` can be visualized in Fig. 27 as a secondary digitizer receiving the edge of the daisy chain signal in close proximity to a reference clock edge (where it is supposed to trigger its data acquisition process). This can occur due to large propagation delay between digitizers, normally caused by long cables or suboptimal topologies. The status is a warning and not an error because it does not prevent the propagation of the daisy chain signal. Once asserted, the daisy chain will continue to operate normally. However, as the delay increases, there will be a point where the edge of the daisy chain signal is pushed into the next reference clock cycle—at which point a warning condition is no longer reported, but the acquired data will be misaligned. The time window (the offset from the reference clock edge) used to monitor this warning scales with the `sampling_frequency`. A high frequency yields a short window and vice versa.

The `rearm_error` is asserted if the primary digitizer's daisy signal propagation module has detected a trigger event during its rearm period, which is static at two reference clock cycles. Note that this is not the same as the rearm time of the full daisy chain, which scales with the record length according to (18) in Section 7.4.6. If asserted, the trigger source is definitely running at an event rate that is too high. However, if cleared, the rearm time of the full daisy chain may still be violated. This will manifest as missing data.

8 Ports

A *port* is a physical interface on the digitizer, excluding the analog inputs (the channels) and the device-to-host interface (e.g. the PCIe board connector). A port may consist of one or several *pins*, depending on the digitizer model. A pin represents an interface for an electrical signal.

A port may offer a unique feature set, a shared feature set or a combination of the two. For example, edge events on both the TRIG and SYNC ports may be used to trigger the data acquisition process, but the timing precision of TRIG events is higher. Thus, high precision edge detection is a unique feature of the TRIG port. However, both ports can be configured to output the digital signal from one of the internal pulse generators—either targeting the same generator, or different generators. Thus, pulse generator output is a shared feature.

A port may also be dedicated to a specific feature, e.g. the CLK port exposes a signal path to the digitizer's clocking system (Section 4). The outline of this section is as follows:

- Section 8.1 describes the *connector map* for each ADQ3 series digitizer. A schematic view of the physical connectors are presented together with tables listing the properties of each connector pin.
- Section 8.2 provides a functional view of a pin using *single-ended* (SE) signaling.
- Section 8.3 provides a functional view of a pin using *differential* signaling.
- Section 8.4 provides a functional view of a *power* pin.
- Section 8.5 provides a functional view of a *clock* pin.
- Section 8.6 provides a description of how to configure the parameters of a pin.

8.1 Connector Map

This section presents the connector map for each ADQ3 series digitizer. Refer to the corresponding subsection for information about the digitizer's physical connectors and how their pins map to the interface provided by the API.

8.1.1 ADQ30-PCIe, ADQ32-PCIe, ADQ33-PCIe

There are four single-pin connectors:

- TRIG, mapping to `ADQ_PORT_TRIG`,
- SYNC, mapping to `ADQ_PORT_SYNC`,
- GPIO, mapping to `ADQ_PORT_GPIOA`;
- CLK, mapping to `ADQ_PORT_CLK`.

Additionally, there is one connector for a flexible flat cable (FFC):

- FFC, housing the two ports: `ADQ_PORT_GPIOB` and `ADQ_PORT_GPIOC`

The mapping between indexes in the port's `pin` array and the connector pins are shown in Fig. 32. Table 8 presents the capabilities of each pin. Refer to the Sections 8.2–8.6 for more information about the pin parameters and how to configure them.

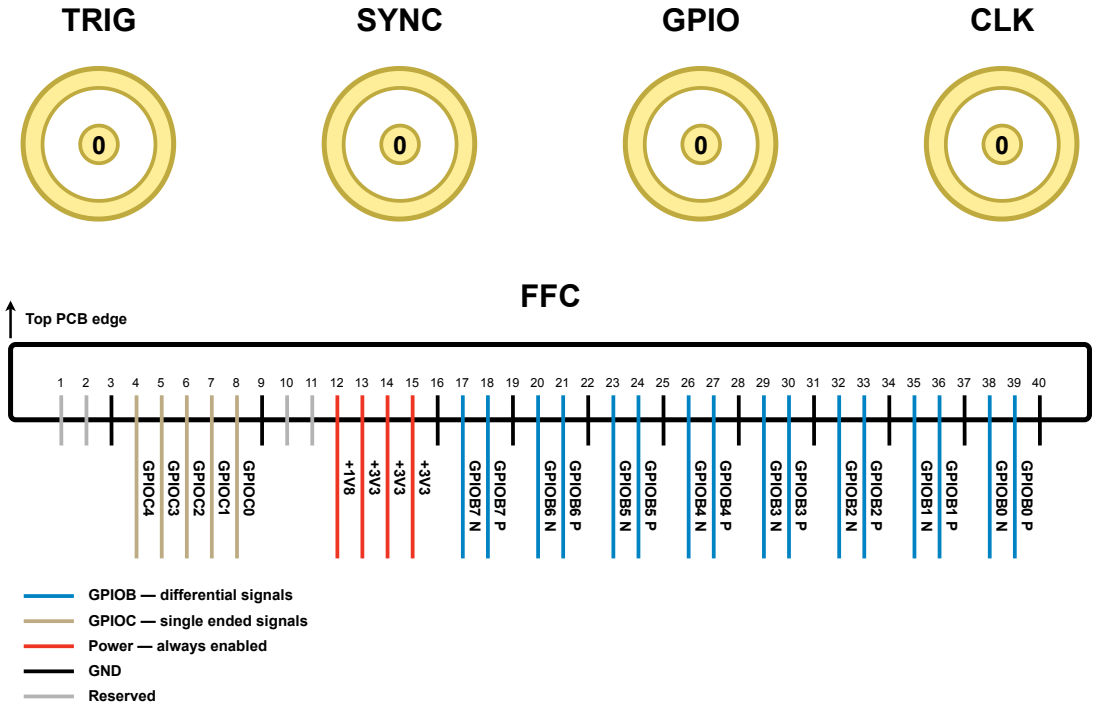


Figure 32: Connector overview for ADQ30-PCIe, ADQ32-PCIe and ADQ33-PCIe.

Port [pin]	ADQ_FUNCTION_PATTERN_GENERATOR0	ADQ_FUNCTION_PATTERN_GENERATOR1	ADQ_FUNCTION_GPIO	ADQ_FUNCTION_PULSE_GENERATOR0	ADQ_FUNCTION_PULSE_GENERATOR1	ADQ_FUNCTION_PULSE_GENERATOR2	ADQ_FUNCTION_PULSE_GENERATOR3	ADQ_FUNCTION_USER_LOGIC	ADQ_FUNCTION_DAISY_CHAIN	Event source	Direction	Input impedance	Type
TRIG[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	50 Ω/ High[1]	Single-ended
SYNC[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	50 Ω/ High[1]	Single-ended
CLK[0]											I/O	50 Ω/ High[1]	Clock
GPIOA[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	50 Ω/ High[1]	Single-ended
GPIOB[0]			✔					✔			I/O	High	Differential
GPIOB[1]			✔					✔			I/O	High	Differential
GPIOB[2]			✔					✔			I/O	High	Differential
GPIOB[3]			✔					✔			I/O	High	Differential
GPIOB[4]			✔					✔			I/O	High	Differential
GPIOB[5]			✔					✔			I/O	High	Differential
GPIOB[6]			✔					✔			I/O	High	Differential
GPIOB[7]			✔					✔			I/O	High	Differential
GPIOC[0]			✔					✔			I/O	High	Single-ended
GPIOC[1]			✔					✔			I/O	High	Single-ended
GPIOC[2]			✔					✔			I/O	High	Single-ended
GPIOC[3]			✔					✔			I/O	High	Single-ended
GPIOC[4]			✔					✔			I/O	High	Single-ended

Table 8: Connector map for ADQ30-PCIe, ADQ32-PCIe and ADQ33-PCIe.

[1] The `input_impedance` is configurable via the corresponding pin parameter.

8.1.2 ADQ36-PXIe

There are three single-pin connectors:

- TRIG, mapping to `ADQ_PORT_TRIG`,
- SYNC, mapping to `ADQ_PORT_SYNC`; and
- CLK, mapping to `ADQ_PORT_CLK`.

Additionally, there are two multi-pin connectors:

- GPIO, housing the three ports: `ADQ_PORT_GPIOA`, `ADQ_PORT_GPIOB` and `ADQ_PORT_GPIOC`; and
- PXIe, mapping to `ADQ_PORT_PXIE`.

The mapping between indexes in the port's `pin` array and the connector pins are shown in Fig. 33. Table 9 presents the capabilities of each pin. Refer to the Sections 8.2–8.6 for more information about the pin parameters and how to configure them.

Example

The GPIOA port contains 12 pins, indexed from 0 to 11 in the corresponding `pin` array. The parameters for `pin[0]` control connector pin 31, `pin[1]` connector pin 32 and so on.

Example

The GPIOB port contains 7 pins using differential signaling, meaning that each port pin involves two connector pins: P and N. The port pins are indexed from 0 to 6 in the corresponding `pin` array where `pin[0]` maps to connector pins 1 and 2.

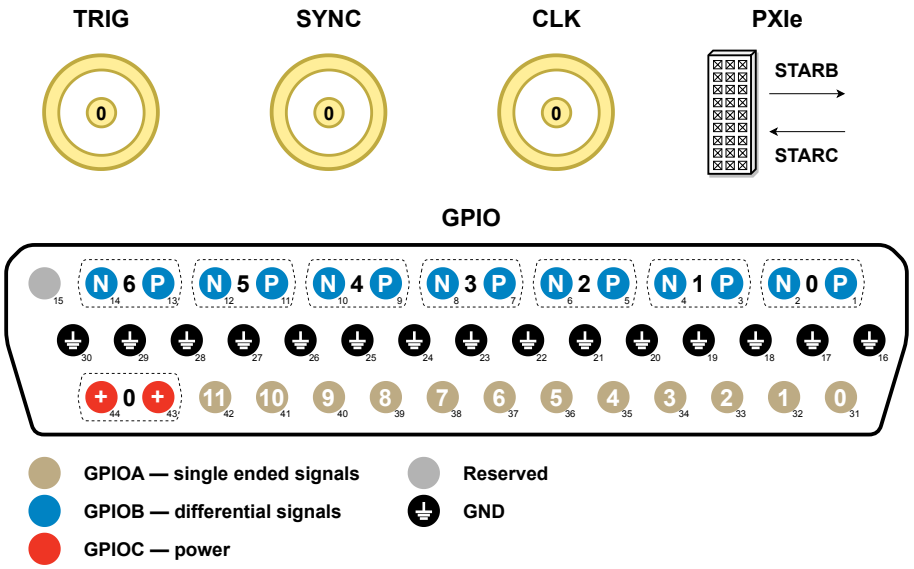


Figure 33: Connector overview for ADQ36-PXIe.

Port [pin]	ADQ_FUNCTION_PATTERN_GENERATOR0	ADQ_FUNCTION_PATTERN_GENERATOR1	ADQ_FUNCTION_GPIO	ADQ_FUNCTION_PULSE_GENERATOR0	ADQ_FUNCTION_PULSE_GENERATOR1	ADQ_FUNCTION_PULSE_GENERATOR2	ADQ_FUNCTION_PULSE_GENERATOR3	ADQ_FUNCTION_USER_LOGIC	ADQ_FUNCTION_DAISY_CHAIN	Event source	Direction	Input impedance	Type
TRIG[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	50 Ω/ High[1]	Single-ended
SYNC[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	50 Ω/ High[1]	Single-ended
CLK[0]											I/O	50 Ω/ High[1]	Clock
GPIOA[0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O[2]	High	Single-ended
GPIOA[1]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[2]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[3]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[4]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[5]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[6]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[7]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[8]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[9]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[10]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended
GPIOA[11]	✔	✔	✔	✔	✔	✔	✔	✔	✔		I/O[2]	High	Single-ended

Table 9: Connector map for ADQ36-PXIe.

[1] The `input_impedance` is configurable via the corresponding pin parameter.

[2] The pin `direction` is controlled in groups of two: pins $2n$ and $2n+1$ must have the same I/O configuration.

Port [pin]	ADQ_FUNCTION_PATTERN_GENERATOR0	ADQ_FUNCTION_PATTERN_GENERATOR1	ADQ_FUNCTION_GPIO	ADQ_FUNCTION_PULSE_GENERATOR0	ADQ_FUNCTION_PULSE_GENERATOR1	ADQ_FUNCTION_PULSE_GENERATOR2	ADQ_FUNCTION_PULSE_GENERATOR3	ADQ_FUNCTION_USER_LOGIC	ADQ_FUNCTION_DAISY_CHAIN	Event source	Direction	Input impedance	Type
GPIOB[0]										✔	I	100 Ω	Differential
GPIOB[1]											I	100 Ω	Differential
GPIOB[2]											I	100 Ω	Differential
GPIOB[3]											I	100 Ω	Differential
GPIOB[4]	✔	✔	✔	✔	✔	✔	✔	✔	✔		O		Differential
GPIOB[5]	✔	✔	✔	✔	✔	✔	✔	✔	✔		O		Differential
GPIOB[6]	✔	✔	✔	✔	✔	✔	✔	✔	✔		O		Differential
GPIOC[0]											O		Power
PXIE[STARB]										✔	I	100 Ω	Differential
PXIE[STARC]	✔	✔	✔	✔	✔	✔	✔	✔	✔		O		Differential
PXIE[TRIG0]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	High	Single-ended
PXIE[TRIG1]	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	I/O	High	Single-ended

Table 9 (continued): Connector map for ADQ36-PXIe.

Classification Public
Revision 2023.3
Document ID 21-2539
Date 2023-11-21

8.2 Single-Ended Signaling

In single-ended signaling, one wire carries a varying voltage and the *signal* is extracted by comparing against a fixed reference voltage. This reference voltage is *ground* for all ADQ3 series digitizers. Fig. 34 presents a functional block diagram of a bidirectional pin using single-ended signaling.

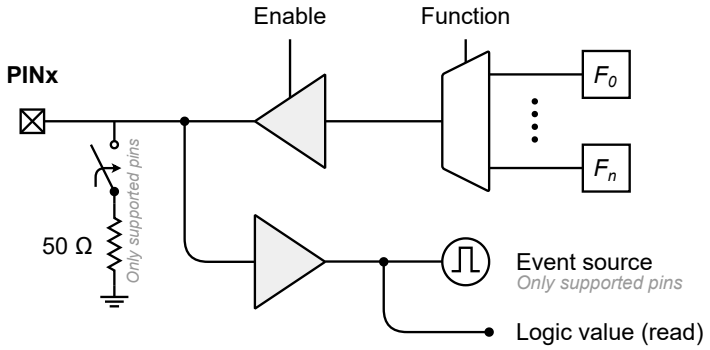


Figure 34: A functional block diagram of a pin using single-ended signaling.

A bidirectional pin defaults to an input, with the output buffer disabled. Some pins have configurable input impedance and some pins have an event source associated with them. If the output buffer is enabled, the digitizer starts driving the output signal from the selected function source (functions are described in Section 7). As an output, the impedance is fixed. The specific capabilities of each pin is listed in the connector map tables in Section 8.1. Refer to Section 8.6 for details on how to configure the pin parameters. Refer to the corresponding product datasheet for pin specifications, e.g. signaling voltages and maximum ratings.

8.3 Differential Signaling

In differential signaling, two wires carry varying voltages and the *signal* is extracted by comparing their voltages. This means that a signal of this type is transmitted, or received, using two physical pins on the digitizer. Figs. 35 and 36 each present a functional block diagram of a unidirectional pin using differential signaling.

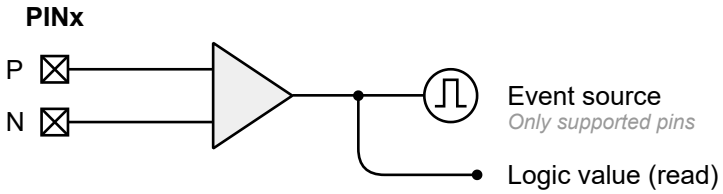


Figure 35: A functional block diagram of an input pin using differential signaling.

The properties of the input and output paths are similar to that of a pin using single-ended signaling (Section 8.2). Some input pins have an associated event source, though the input impedance is not configurable for this type of pin. The specific capabilities of each pin is listed in the connector map tables

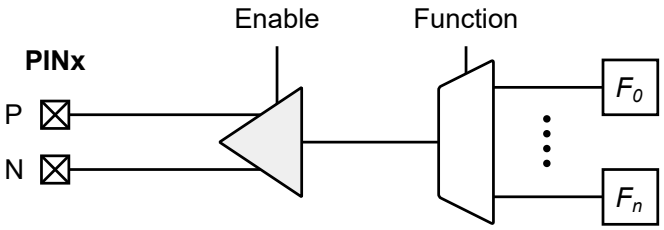


Figure 36: A functional block diagram of an output pin using differential signaling.

in Section 8.1. Refer to Section 8.6 for details on how to configure the pin parameters. Refer to the corresponding product datasheet for pin specifications, e.g. signaling voltages and maximum ratings.

Important

Event sources associated with a differential input may be *unstable* if left unconnected. When the signals are floating, events may be generated at random. Make sure to only use these event sources when the input is driven in accordance with the datasheet specification. [1] [2] [3] [4]

8.4 Power

A power pin is capable of capable of providing a current at a constant voltage to power external electrical circuits. The output current is limited by the digitizer so as to not damage the power supply. Fig. 37 presents a functional block diagram of a power pin.

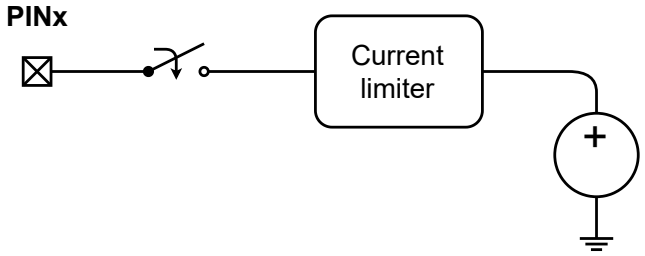


Figure 37: A functional block diagram of a pin capable of supplying power to external circuitry.

In terms of configuration, a power pin only offers an enable/disable mechanism. By default, the pin is disabled, disconnecting the power supply. Setting the pin parameter `value` to 1 will enable the pin and connect the power supply. Returning the parameter to 0 will disable the pin. Refer to Section 8.6 for details on how to configure the pin parameters. Refer to the corresponding product datasheet for pin specifications, e.g. output voltage and maximum current.

8.5 Clock

A clock pin features dedicated routing to the digitizer's clock system. The purpose is to either receive a clock signal using the input path, or to use the output path to transmit a clock signal, making the digitizer into a reference for other external devices. Refer to Section 4 for more information about the clock system. Fig. 38 presents a functional block diagram of a clock pin.

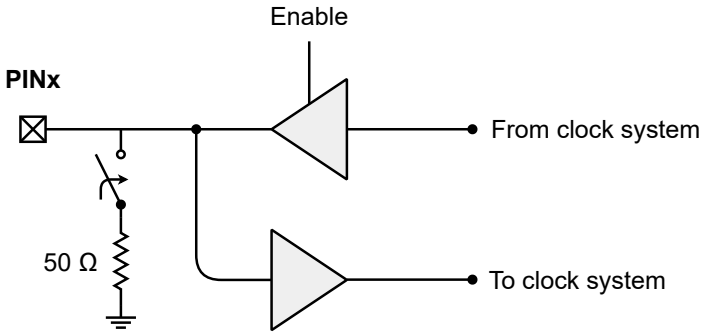


Figure 38: A functional block diagram of a pin with dedicated routing to the digitizer's clock system.

A clock pin allows configuration of the port direction as well as the input impedance. Refer to Section 8.6 for details on how to configure the pin parameters and to the corresponding product datasheet for pin specifications, e.g. signaling voltages and maximum ratings.

8.6 Pin Configuration

The properties of a pin is configured as part of the parameter set for the enclosing port. This means that while the parameters of a pin may be controlled individually, the parameters are updated for all the pins in a port at the same time.

For each port, the pin parameters are organized into an array with `ADQ_MAX_NOF_PINS` entries. However, only a subset of these may be *active* entries, i.e. mapping to an actual pin in the port. The parameters for inactive entries are ignored. The number of active entries in the `pin` array is determined by the value of the port's corresponding constant parameter `nof_pins`. In general, the array indexing starts at zero, meaning that the parameters of the first pin in the port are given by `pin[0]`. The exception to this rule are the PXIe and MTCA ports, for which the pins are *named* and preferably accessed as `pin[ADQ_PIN_PXIE_STARB]`.

A pin is configured as an input by assigning the value `ADQ_DIRECTION_IN` to the parameter `direction`. Additionally, if the pin supports it, the parameter `input_impedance` may be set to an appropriate value from the enumeration `ADQImpedance` to modify the input impedance.

A pin is configured as an output by assigning the value `ADQ_DIRECTION_OUT` to the parameter `direction`. This enables the output buffer. Since the pin must always output a well-defined signal, a `function` must be selected from those supported by the pin. The output signal can be digitally inverted by setting `invert_output` to a nonzero value.

Note

For bidirectional pins, the input path is always active, allowing the digitizer to monitor its own output.

The pin parameter `value` behaves differently depending on the API function operating on the parameter set:

- If the pin has an input path, a call to `GetParameters()` will set `value` to reflect the digital signal level currently at the pin. The timing of this operation is *unknown*, i.e. consecutive calls cannot be

used to sample the input pin with any guaranteed precision. This operation is not supported for pins without an input path.

- If the pin is configured as an output, a call to `SetParameters()` together with the `function` set to `ADQ_FUNCTION_GPIO`, will set the digital signal level of the pin to `value`. This operation is not supported for pins without an output path.

8.6.1 Example: Pattern Generator Output

The code snippet below sets up the TRIG port as an output and selects the first pattern generator (Section 7.1) as the source for the output signal.

```
/* Initialize the parameters for the TRIG port to their default values. */
struct ADQPortParameters port_trig;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_PORT_TRIG, &port_trig);
if (result != sizeof(port_trig))
{
  /* Handle error */
}

/* Enable the output buffer and select the output of pattern generator 0. */
port_trig.pin[0].function = ADQ_FUNCTION_PATTERN_GENERATOR0;
port_trig.pin[0].direction = ADQ_DIRECTION_OUT;
port_trig.pin[0].invert_output = 0;

/* Set the port parameters. */
result = ADQ_SetParameters(adq_cu, adq_num, &port_trig);
if (result != sizeof(port_trig))
{
  /* Handle error */
}
```

8.6.2 Example: Pulse Generator Output

The code snippet below sets up the SYNC port as an output and selects the first pulse generator (Section 7.2) as the source for the output signal. The pulse generator is configured to *follow* the periodic event source to create signal with 50% duty cycle with the same frequency as the event source.

This particular code snippet uses the top-level parameter struct to configure all the relevant parts of the system with one single call.

Important

The code below will result in all the untouched parameters being set to their *default* values due to the call to `InitializeParameters()`. These calls are meant to provide context for the parameter assignments needed to accomplish this particular task.

```
/* Initialize the parameters for the TRIG port to their default values. */
struct ADQParameters adq;
int result = ADQ_InitializeParameters(adq_cu, adq_num, ADQ_PARAMETER_ID_TOP, &adq);
if (result != sizeof(adq))
{
  /* Handle error */
}

/* Configure the periodic event source to run at 1 kHz. */
adq.event_source.periodic.frequency = 1000.0;

/* Configure the pulse generator to 'follow' the periodic event source,
   yielding a duty cycle of 50%. */
adq.function.pulse_generator[0].source = ADQ_EVENT_SOURCE_PERIODIC;
adq.function.pulse_generator[0].edge = ADQ_EDGE_RISING;
adq.function.pulse_generator[0].length = -1;

/* Enable the output buffer and select the output of pulse generator 0. */
adq.port[ADQ_PORT_SYNC].pin[0].function = ADQ_FUNCTION_PULSE_GENERATOR0;
adq.port[ADQ_PORT_SYNC].pin[0].direction = ADQ_DIRECTION_OUT;
adq.port[ADQ_PORT_SYNC].pin[0].invert_output = 0;

/* Set the port parameters. */
result = ADQ_SetParameters(adq_cu, adq_num, &adq);
if (result != sizeof(adq))
{
  /* Handle error */
}
```

8.6.3 Example: Software Controlled GPIO

The code snippet below demonstrates software controlled GPIO of pins 0 and 1 in the GPIOA port. Note that not all ADQ3 series digitizers have two pins in the GPIOA port.

```
/* Initialize the parameters of the GPIOA port to their default values. */
struct ADQPortParameters port_gpioa;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_PORT_GPIOA, &port_gpioa);
if (result != sizeof(port_gpioa))
{
  /* Handle error */
}

/* Configure GPIOA0 as a software controlled output and
   keep the other pins as inputs. */
port_gpioa.pin[0].function = ADQ_FUNCTION_GPIO;
port_gpioa.pin[0].direction = ADQ_DIRECTION_OUT;
port_gpioa.pin[0].invert_output = 0;
port_gpioa.pin[0].value = 1;

/* Set the port parameters. GPIOA0 will transition to logic high. */
result = ADQ_SetParameters(adq_cu, adq_num, &port_gpioa);
if (result != sizeof(port_gpioa))
{
  /* Handle error */
}

/* Get the current parameters, sampling all pins with an input path. */
result = ADQ_GetParameters(adq_cu, adq_num,
                           ADQ_PARAMETER_ID_PORT_GPIOA, &port_gpioa);
if (result != sizeof(port_gpioa))
{
  /* Handle error */
}

/* Use the logic level of of GPIOA1 to direct the control flow. */
if (port_gpioa.pin[1].value != 0)
  /* GPIOA1 is logic high. */
else
  /* GPIOA1 is logic low. */

/* Set GPIOA0 to logic low. */
port_gpioa.pin[0].value = 0;
result = ADQ_SetParameters(adq_cu, adq_num, &port_gpioa);
if (result != sizeof(port_gpioa))
{
  /* Handle error */
}
```

8.6.4 Example: Reference Clock Output

The code snippet below demonstrates how to output the internal reference clock on the CLK port.

```
/* Initialize the parameters of the CLK port to their default values. */
struct ADQPortParameters port_clk;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_PORT_CLK, &port_clk);
if (result != sizeof(port_clk))
{
  /* Handle error */
}

/* Configure CLK to output the reference clock. */
port_clk.pin[0].direction = ADQ_DIRECTION_OUT;

/* Set the port parameters. */
result = ADQ_SetParameters(adq_cu, adq_num, &port_clk);
if (result != sizeof(port_clk))
{
  /* Handle error */
}
```

9 Data Acquisition

Data acquisition is the process of extracting *records* from the ADC data stream on a trigger event and storing this data in the digitizer's on-board memory. This memory acts as a buffer for the physical interface (PCIe).

Note

The digitizer's on-board memory acts as a buffer for the physical interface.

Important

Data acquisition is only possible when the digitizer's licenses fulfill the requirements of the active firmware. See Section 12.2 for more information.

The core acquisition parameters are:

- the number of records to acquire,
- the record length, expressed either
  - as a static value; or
  - as constraints for a dynamic record (see Section 9.1),
- the trigger event source and the edge sensitivity (see Section 6),
- the horizontal offset; and
- the trigger blocking source (see Section 9.5).

These are members of the parameter set `ADQDataAcquisitionParameters` and are independently configurable for each digitizer `channel`. A channel is considered *active* if the number of records to acquire (`nof_records`) is set to a nonzero value. The special value `ADQ_INFINITE_NOF_RECORDS` may be used to specify an infinite acquisition.

For each active channel, the configuration differs slightly depending on if the length of a record should be *static* (default) or *dynamic*. Configuring the acquisition for records with static length is straightforward: set the `record_length` to the desired length in samples. Additionally, the special value `ADQ_INFINITE_RECORD_LENGTH` is also allowed and signifies the acquisition of a record that never ends (infinite length). This acquisition mode places additional constraints on the data transfer and readout processes, see Section 10.5.7 for details. There is more to consider when configuring records with dynamic length. Refer to Section 9.1 for a detailed description of this acquisition behavior and the controlling parameters.

The `trigger_source` is expected to be set to one of the digitizer's event sources. The trigger source also needs to support the selected edge sensitivity `trigger_edge`.

The `horizontal_offset` shifts the region of captured data in relation to the trigger event. Fig. 39 presents how a record of a constant length is acquired for three values of the horizontal offset:

(a) A negative horizontal offset shifts the region captured as a record to an *earlier* point in time, relative to the trigger event. This is sometimes known as *pretrigger*.

(b) A horizontal offset of zero performs no shift.

(c) A horizontal offset greater than zero shifts the region captured as a record to a *later* point in time. This is sometimes known as *trigger delay*.

Note

The trigger delay mechanism is implemented as an event queue with a maximum capacity of 512 events. This means that if 512 trigger events are observed before the first entry is ejected (the queue is filled to capacity within the specified trigger delay), the 513th event will not be detected. This is rarely a problem in practice.

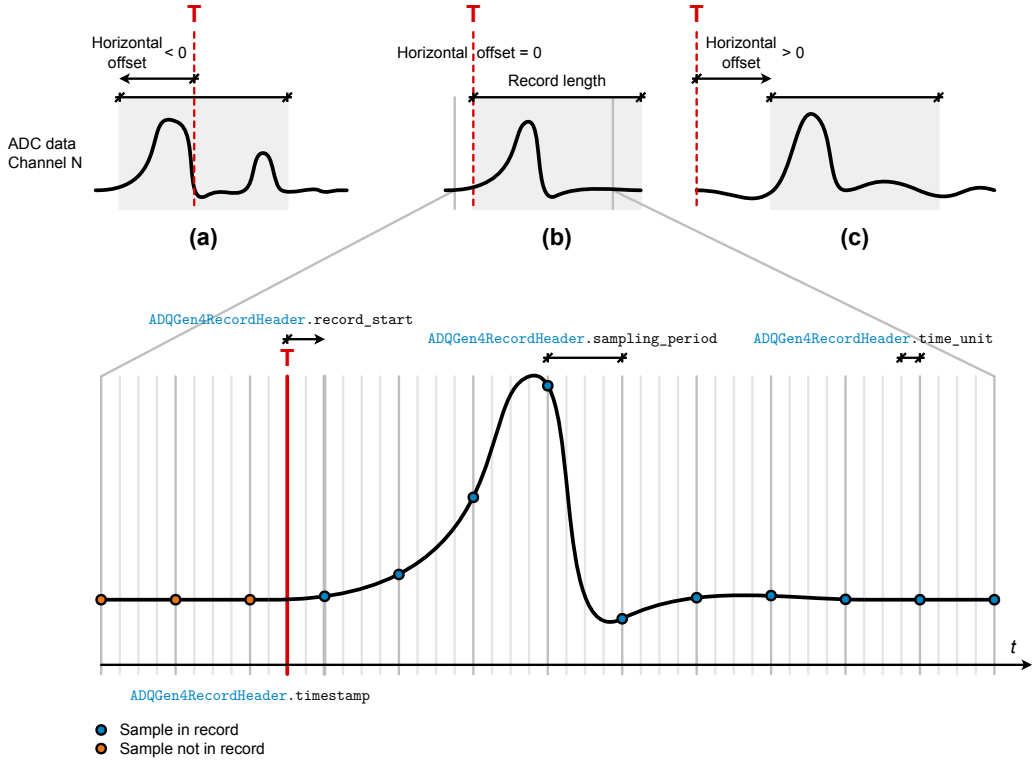


Figure 39: Illustration of how the parameters of the data acquisition process affect the acquisition of a record (the shaded region). Relative values, e.g. the record header field `record_start`, are all anchored in the trigger event **T**. The header field `timestamp` specifies where the trigger event is located on the timing grid.

9.1 Dynamic Record Length

The digitizer can be configured to acquire records with dynamic length. Compared to records with a static length, these records have their length determined in real time by the data acquisition process. This feature is useful for sophisticated data rate reduction in cases where the region of interest in the ADC data stream may vary in size.

For example, applications measuring pulses with varying shape can have a record fit a pulse exactly, rather than having to define a static record length large enough to capture the worst-case pulse. The drawback of the latter method is that a long record framing a short pulse will contain a large “silent” part,

i.e. data that is not interesting but has been transferred to the host computer nonetheless. Discarding these regions in the acquisition process is often called *zero suppression*. See Section 9.1.3 for additional details.

Another acquisition behavior that can be realized using the dynamic length mechanism is *gated acquisition*. In such a system, an external digital signal, e.g. input on the TRIG connector, dynamically defines a region of interest which should be acquired as a single continuous record. Refer to Section 9.1.4 for an example.

By default, the digitizer is configured to acquire records with static length. To change the acquisition behavior, set `dynamic_record_length_enabled` to a nonzero value. The events from the selected `trigger_source` will define the dynamic part of the record starting from an event of the specified `trigger_edge` and extending to its complementary (opposite) edge event. The selection of the complementary event is automatic and cannot be chosen independently by the user. For example, if `trigger_edge` is set to `ADQ_EDGE_RISING`, the complementary event is `ADQ_EDGE_FALLING` of the same `trigger_source` and vice versa. For the sake of simplicity, the `trigger_edge` cannot be set to `ADQ_EDGE_BOTH` in this acquisition mode.

Acquiring records with dynamic length places additional requirements on the data transfer process, since some simplifying assumptions made for records with static length no longer hold true. Refer to Section 10.5.3 for additional details on how to configure the data transfer process to be able to transfer records with dynamic length.

Important

To transfer records with dynamic length, the data transfer process has to be configured accordingly. Refer to Section 10.5.3 for detailed instructions.

9.1.1 Edge Windows

A record with dynamic length consists of three parts:

- the leading edge window (LEW),
- the dynamic window; and
- the trailing edge window (TEW),

where the two edge windows are static in length and only the dynamic window may extend arbitrarily. The length of the two edge windows must be decided at the time of configuration and is specified by the parameters `dynamic_leading_edge_window_length` and `dynamic_trailing_edge_window_length`. The length of the leading edge window may be set to zero but the length of the trailing edge window must always be set to a nonzero value since the record must fulfill the requirements on minimum record length as the dynamic window length approaches zero. Fig. 40 presents the anatomy of a record with dynamic length where the `trigger_edge` is set to `ADQ_EDGE_RISING`.

9.1.2 Overlap and Maximum Length

Since the leading edge window length is allowed to be greater than the rearm time of the digitizer, consecutive records can *overlap*, defined as the leading edge window of a record including the trailing edge

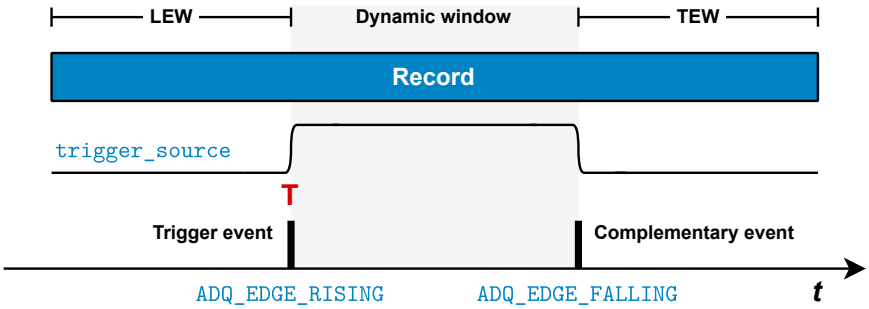


Figure 40: The anatomy of a record with dynamic length. The `trigger_edge` is set to `ADQ_EDGE_RISING`, automatically selecting the complementary event `ADQ_EDGE_FALLING` as the stop condition for the dynamic window. This region is flanked by two windows of static length, the leading edge window (LEW) and the trailing edge window (TEW).

window of the previous record. When this occurs, the first record is *extended* with the contents of the second record, as if its original end condition was never met. If a third record also overlaps, the extension process continues.

The rationale for this behavior is as follows: the dynamic length mechanism exists to allow real-time events to define the length of the region of interest within the ADC data stream. As such, when the user-defined edge windows combines with trigger events in close enough proximity to trigger an overlap, the overlapping region is interpreted as *twice* as interesting. However, ADC data samples cannot belong to two separate records at the same time, creating a conflict which is resolved by creating a longer record.

Obviously, this can lead to an unintended situation where a high trigger rate causes a (for practical purposes) “infinite” record, overflowing the device-to-host interface or overwhelming any subsequent processing logic. For this reason, it is possible to configure a maximum record length via the parameter `dynamic_record_length_max`. When a record reaches this limit, it is forcefully cut short. By default, the limit is set to the special value `ADQ_INFINITE_RECORD_LENGTH`, allowing the record to grow indefinitely. This limit is applied regardless of whether the record is extended or not.

Fig. 41 shows an example where two trigger events occur in such close proximity that their regions of interest overlap, extending the record until the maximum length is reached, at which point the record is cut short. The record that propagates to the user will report the trigger event **T** in the record `header` and the trigger event that caused the extension will be suppressed.

9.1.3 Zero Suppression for Unipolar Pulse Data

Fig. 42 presents the concept of using the dynamic record length mechanism to implement zero suppression for unipolar pulse data. The configuration centers around selecting `ADQ_EVENT_SOURCE_LEVEL` as the channel’s `trigger_source`. Other channels may be triggered with the same acquisition pattern by targeting the signal level event source for a specific channel, e.g. `ADQ_EVENT_SOURCE_LEVEL_CHANNEL0`. The `trigger_edge` should be set according to the polarity of the pulses, i.e.

- `ADQ_EDGE_RISING` for pulses with positive polarity; and
- `ADQ_EDGE_FALLING` for pulses with negative polarity.

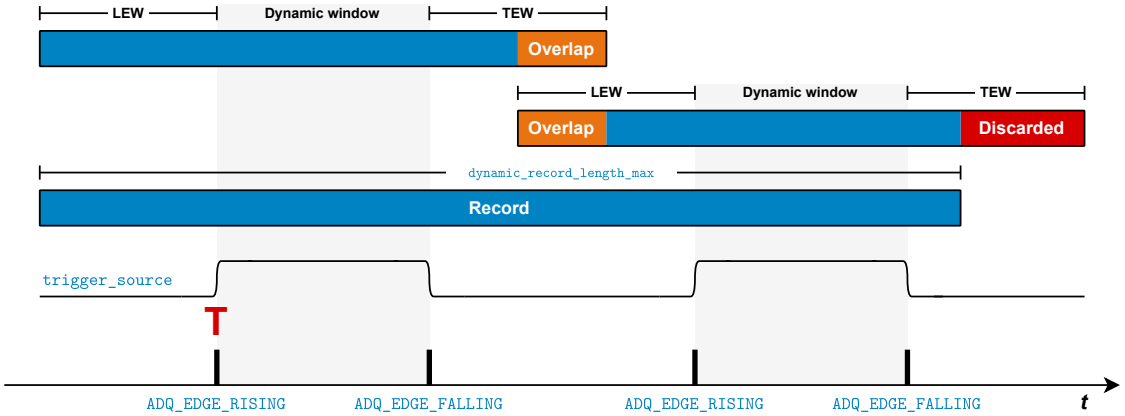


Figure 41: An overlap is resolved by extending the record, either until it ends naturally by reaching the end of the trailing edge window defined by the extension event, or until the maximum length specified by `dynamic_record_length_max` is reached.

This will automatically select the complementary event as the point where the `level` is crossed in the opposite direction—thus making the record length scale with the pulse width.

The data for regions with a high concentration of pulses (bursts) may end up in the same record, according to the overlap rules detailed in Section 9.1.2. When this occurs, the record's `timestamp` will point to the trigger event of the first pulse in the burst.

This use case benefits significantly from using the digital baseline stabilization (DBS) signal processing module to ensure a stable baseline with minimal drift. Refer to Section 5.3 for more information.

Optionally, the trigger blocking mechanism (Section 9.5) can be used to implement a *detection window*. This window acts as a filter on top of the behavior presented in Fig. 42. A detection window may be defined and controlled by external events and records can only be acquired while the window is open. This case is explained in more detail in the context of the FWPD firmware in Section 5.7.8.

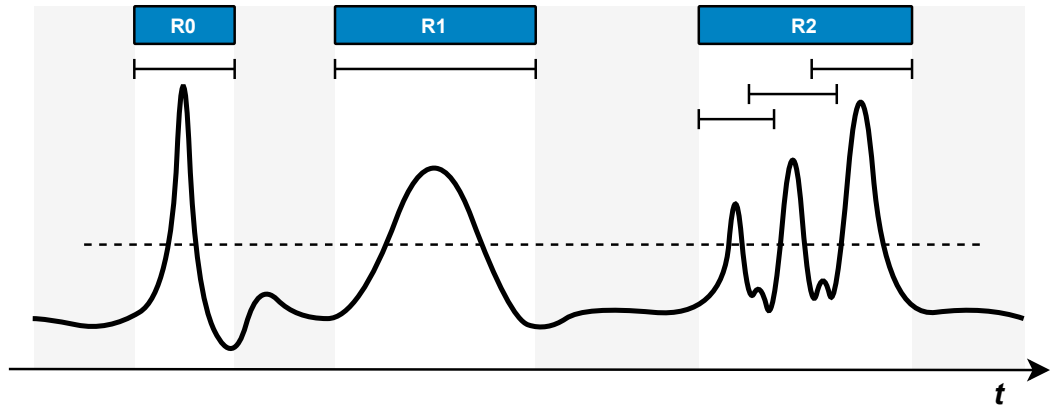


Figure 42: An example of using the dynamic record length mechanism to realize zero suppression for pulse data. The `trigger_source` is set to `ADQ_EVENT_SOURCE_LEVEL`, making the record length scale with the width of a pulse. A burst of pulses may end up in the same record if their regions of interest overlap, as described in Section 9.1.2.

9.1.4 Gated Acquisition

Fig. 43 shows the concept of using the dynamic record length mechanism to implement gated acquisition. The configuration centers around using an external signal with two levels: logic high and logic low. The digitizer should acquire data as long as the signal remains either logic high or logic low, depending on the polarity. In Fig. 43, logic high defines the gate so the `trigger_edge` is set to `ADQ_EDGE_RISING`. The signal is connected to the TRIG port, thus `trigger_source` should be set to `ADQ_EVENT_SOURCE_TRIG`.

By default `horizontal_offset` is zero, meaning that the start of the record coincides with the trigger event **T**. Applying a horizontal offset will shift the acquisition region relative to the trigger. In Fig. 43, a positive `horizontal_offset` shifts to the right, *delaying* the record while still keeping its length defined by the gate. Conversely, a negative horizontal offset shifts in the other direction. However, the range is limited such that the sum

$$|\texttt{horizontal\_offset}| + \texttt{dynamic\_leading\_edge\_window\_length}$$

cannot exceed a certain value. Refer to the parameter documentation for details.

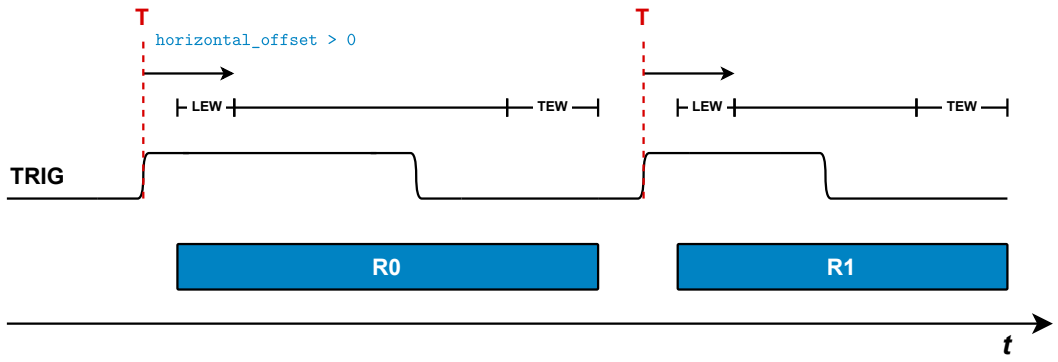


Figure 43: An example of using the dynamic record length mechanism to realize gated acquisition. The gate is defined by a logic high level of the signal connected to the TRIG port. Thus, the `trigger_source` should be set to `ADQ_EVENT_SOURCE_TRIG` and `trigger_edge` to `ADQ_EDGE_RISING`. A `horizontal_offset` > 0 has been applied, causing the entire acquisition region to shift to the right.

9.2 Rearm Time

The digitizer enforces a minimum time between records, i.e. a window following the end of a record where trigger events are ignored. This is called the *rearm time* of the data acquisition process and is specified in the digitizer's datasheet [1] [2] [3] [4] and is normally in the range of a few tens of nanoseconds. Through the parameter `rearm_length`, it is possible to extend the length of this rearm window. While this is not normally desired, it offers a simple way to ensure a maximum trigger rate. The default value is set to the minimum allowed value, as specified in the datasheet.

Example

Consider an oscilloscope-like GUI application where the digitizer is configured to create an infinte stream of records and a graph is continuously updated to show the latest acquired record. In practice, displays have a maximum refresh rate, effectively limiting the rate at which information can be visualized. This directly translates into a maximum trigger rate if the goal is to display *every* record. Otherwise, some records will need to be discarded as they would end up between frames.

Consider a refresh rate of 60 Hz and a digitizer with a sampling rate of 2500 MSPS. A pessimistic value for the length of the rearm window that ensures a maximum trigger rate of 60 Hz is

$$\frac{2500 \cdot 10^6}{60} = 41666666.67 \approx 41666667$$

which can be specified as the new `rearm_length` after ceiling the value to a multiple of the firmware-specific step size. If the digitizer is configured to acquire records of static length, the value can be refined as

$$\frac{2500 \cdot 10^6}{60} - \texttt{record\_length}.$$

9.3 Timing Information

The digitizer has a built-in time-keeping mechanism. At power on, a counter starts to monotonically increment—creating a timing grid that is in phase with the sampling grid. In addition to the acquired ADC data, the digitizer keeps track of the record's *timestamp* and a value called *record start*. The timestamp specifies where the trigger event is located on the timing grid. The record start value specifies where the first sample in the record is located, relative to the timestamp. These values are propagated to the user application via the record header as the two fields `timestamp` and `record_start`. Thus, the timestamp of the *first sample* in the record, $x[0]$, is calculated as

$$t_{x[0]} = \texttt{timestamp} + \texttt{record\_start}. \tag{27}$$

The header field `time_unit` specifies the value of one timing grid unit in seconds. The other horizontal header fields, i.e. `record_start`, `timestamp` and `sampling_period`, are expressed as an integer number of time units.

Note that the timing resolution varies between event sources. For example, a signal level event will have sample resolution while an event from the TRIG port will have subsample resolution. Refer to Section 6 for a description of the event sources and to the product datasheet for the information on timing resolution [1] [2] [3] [4].

Important

The timing resolution varies between event sources.

9.3.1 Floating Point Inaccuracies

The `time_unit` is subject to the general intricacies of floating point numbers. It is sufficiently precise to fulfill its intended purpose: to specify the value of one timing grid unit in seconds. However, care must be taken when this value is used for other tasks, e.g. calculating the absolute point in time of a sample.

Floating point numbers are useful because they can be made accurate enough for *practical purposes*. However, there is always a trade-off between range and resolution. This is particularly important in the data acquisition context because a digitizer can output a record with a large number of samples (wide range), where each sample is taken at comparatively minute intervals (high resolution). A general expression for the absolute time of the sample at time instance n can be formulated as

$$t_{x[n],absolute} = \underbrace{(\texttt{timestamp} + \texttt{record\_start} + n \cdot \texttt{sampling\_period})}_{\text{Timing grid position (an integer)}} \cdot \texttt{time\_unit}. \tag{28}$$

As n increases, so do the requirements on the *range* of the floating point value $t_{x[n],absolute}$, while the requirements on the *precision* remain the same. An error in the `time_unit` will scale linearly with n, causing a potentially significant deviation from the *true* absolute position of $x[n]$ if (28) is used to calculate the value. It is up to the user to decide how to represent the sampling points in a way that is sufficient for the target application.

Example

Consider a digitizer running at 2.0 GSPS with a timing grid resolution of 8 times the sampling rate. Using the ubiquitous 64-bit IEEE-754 floating point representation, the `time_unit` written to 32 decimal places is

0.00000000006250000000000000389260 s

This is close enough to the ideal value of 62.5 ps to be useful in practice. However, the value contains an error component δ. Applying (28) to determine the absolute time of sample n will magnify this error to $n\delta$, which *could* have implications for the target application.

This error is also magnified when attempting to derive the sampling frequency as

$$f_s = \frac{1}{\texttt{sampling\_period} \cdot \texttt{time\_unit}}$$

which in this case yields

$$f_s = \frac{1}{8 \cdot 0.0000000000625...} = 1999999999.99999976158... \text{ Hz}$$

which only gives the correct value of 2 GHz after rounding.

9.4 Starting and Stopping

Once the acquisition parameters have been set, the data acquisition process is started by calling `StartDataAcquisition()`. This effectively arms the digitizer, which immediately begins acquiring records on all active channels and writing the resulting data to the on-board memory. Making the data available to the user application is the task of the data transfer and data readout processes, outlined in Section 10.

The start of these processes are also controlled by `StartDataAcquisition()`.

Once the digitizer enters the acquisition phase, it will no longer accept changes to its parameters until the acquisition is stopped. Aborting the acquisition can be done at any time by calling `StopDataAcquisition()`.

Important

The digitizer will not accept changes to its parameters while the data acquisition process is running.

9.5 Trigger Blocking

The trigger blocking feature is used to prevent records from being generated by masking trigger events. Trigger blocking is activated on a per-channel basis by setting the `trigger_blocking_source` to a supported function. The data acquisition process will

- block trigger events while the output of this function is logic high (the *block* state); and
- accept trigger events while the output of this function is logic low (the *accept* state).

Trigger blocking is disabled by setting `trigger_blocking_source` to `ADQ_FUNCTION_INVALID`. This is the default value. Currently, only the pattern generators, described in Section 7.1, can be used as trigger blocking sources.

9.5.1 Zero Length Records

The data acquisition process can be configured to emit a record with *no data*, i.e. with zero length, when the trigger blocking mechanism transitions from the accept state to the block state without having observed any trigger events. Fig. 44 shows the principle of operation.

The purpose of a record with this property is to signal *the absence of data*. This is useful in acquisition systems running on a repetitive schedule where—whether or not an acquisition window is empty—is valuable information that should propagate to the user.

The mechanism is directly connected to the signal used for trigger blocking and thus controlled by the channel's `trigger_blocking_source`. The mechanism is disabled by default and may be enabled by setting the channel-specific parameter `zero_length_records_enabled` to 1. A zero length record only consists of a `header` and may thus only be enabled when metadata is allowed to propagate through the data transfer layer (Section 10), as determined by the parameter `metadata_enabled`. See Section 10.5.5 for more information about how these records are received by the user application.

Since a zero length record is not created by a trigger event and contains no data, the timing information reported in its `header` will have limited usefulness. Specifically, `record_start` is invalid and the `timestamp` will only provide a rough estimate of when this record was created relative to the timing grid. It will be accurate enough for most purposes, such as sorting records chronologically or inferring the shape of the signal from the `trigger_blocking_source`. Its precision is in the range of a few sampling periods. Other `header` values are valid. For example, if the digitizer is configured to synchronize its timestamp upon entering the accept state (Section 7.3), the `timestamp_synchronization_counter` will correctly reflect this increment, allowing the user to identify which window in the sequence that was empty of trigger events.

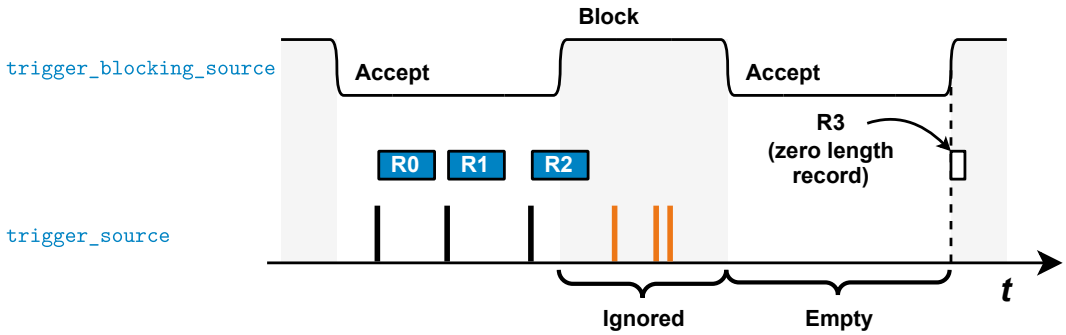


Figure 44: If enabled via `zero_length_records_enabled`, a zero length record is emitted if the trigger blocking mechanism transitions from the accept state to the block state without having observed any trigger events.

10 Data Transfer and Data Readout

Once data has been acquired and stored in the on-board memory, it passes into the domain of the data transfer process. This process moves the data across the physical interface to *transfer buffers* located in the memory of an *endpoint*. An endpoint may be the host computer or a GPU. The destination is determined by the address configuration of the data transfer process.

If the transfer buffers are located in the host computer's RAM, the data readout process is available as an optional layer. With this layer, the data transfer process is managed by a thread which is active as long as an acquisition is ongoing. Records are passed to the user application through thread-safe channels with a bidirectional queue interface to allow reusing memory in an efficient manner. Unless the use case involves advanced requirements, it is recommended to begin developing the user application by using the data readout interface described in Section 10.5. Fig. 45 presents an overview and the following sections describe the processes in more detail.

Important

The data readout process, where records are passed to the user application through thread-safe channels, is only available when the digitizer transfers data to the host computer's RAM, i.e. not when the endpoint is a GPU.

Note

Unless the use case involves advanced requirements, it is recommended to begin developing the user application by using the data readout interface described in Section 10.5.

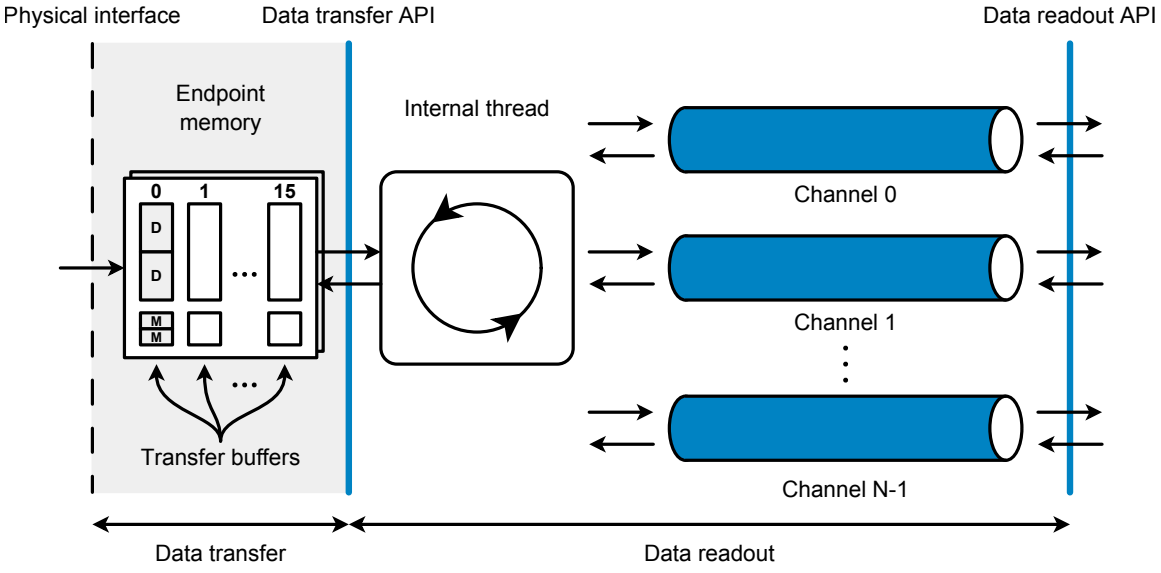


Figure 45: Overview of the data transfer and data readout processes. Data is transferred over the physical interface to transfer buffers in the endpoint memory. If the transfer buffers reside in the host computer's RAM, the data readout process is available as an optional layer. In this case, an internal thread manages the transfer of data between the digitizer and the host computer. Records are passed to the user via thread-safe channels corresponding to channels of the digitizer.

10.1 Transfer Buffers

The data from the digitizer's on-board memory is transferred to the target endpoint and stored in *transfer buffers* before being passed on to the user application via one of the two available interfaces. There are two types of transfer buffers:

- record data transfer buffers; and
- metadata transfer buffers, holding the data that will become record headers.

These are always paired, meaning that each record data transfer buffer have a corresponding metadata transfer buffer and vice versa. Record data is always transferred to the endpoint. Whether or not metadata is transferred is controlled by the channel-specific parameter `metadata_enabled`.

The parameter `nof_buffers` sets the number of *transfer buffer pairs* for a target channel. The value 0 implies that the channel is disabled. No data will be transferred for disabled channels, regardless of the acquisition parameters.

When `transfer_records_to_host_enabled` is set to 1, transfer buffers will be allocated in the host computer's RAM by the API. The size of the transfer buffers affect the throughput with the general rule that throughput increases with size. Typically, a buffer size in the lower MiB range is sufficient to achieve the specified maximum throughput.

Transfer buffers need to be *contiguous*, meaning that they span a single *unbroken* block of memory in the target endpoint. In the case where transfer buffers reside in the host computer's RAM, the allocation can fail due to fragmentation if the requested memory region is too large. In practice, it is recommended to use transfer buffers in the lower MiB range and to transfer large records (exceeding the size of a transfer buffer) in several segments. See Section 10.5.7 for additional details. A transfer buffer should *not* be set in the GiB range under normal circumstances.

Important

A transfer buffer should *not* be set in the GiB range under normal circumstances. Each transfer buffer requires a *contiguous* memory region. Thus, if the size is large, the allocation can fail if the memory has become fragmented.

10.1.1 Advanced Parameters

This sections contains descriptions of advanced transfer buffer parameters and use cases. If the user application reads data via the data readout interface (Section 10.5), these parameters can be ignored.

Important

The parameters outlined in this section can be ignored (use the default values) if the data readout interface is used.

Packed Transfer Buffers

If `packed_buffers_enabled` is set to 1, the API will allocate `nof_buffers` contiguous memory ranges each corresponding to a record transfer buffer index. Each contiguous memory range will contain one record transfer buffer for each active channel. If metadata is enabled, metadata buffers will be allocated in the same manner. This allocation scheme is useful in multichannel applications with high throughput

and small buffer size, where the transferred data is copied to another location like a disk or a GPU. For a given transfer buffer index, data from all active channels can be copied from memory with a single operation, reducing overhead. Certain setup criteria has to be met before this option can be enabled, refer to the parameter documentation for `packed_buffers_enabled` for details.

User Allocated Transfer Buffers

The digitizer can target user supplied transfer buffers by setting `transfer_records_to_host_enabled` to 0 and setting the buffer bus addresses in `record_buffer_bus_address` and `metadata_buffer_bus_address` if metadata is active. Each transfer buffer must be contiguous and available for direct memory access (DMA). User supplied transfer buffers can reside in any or multiple endpoints, including host system RAM. The transfer buffers will always be written in strict order (sequentially), i.e. buffer 1 will always be written after buffer 0.

10.2 Marker Buffers

Each transfer buffer pair has a corresponding *marker buffer* whose purpose is to indicate the status of the transfer buffer pair, i.e. whether or not new data is available. For most use cases, markers are handled by the API and are never encountered by the user application. This is the case for the data readout interface (Section 10.5) where `marker_mode` is set to `ADQ_MARKER_MODE_HOST_AUTO`.

Important

Marker buffers are handled by the API, out of sight from the user when the data readout interface is used (Section 10.5).

More advanced use cases may require manual handling of the marker buffers. These are outlined in Section 10.2.1.

10.2.1 Advanced Use Cases

The marker buffer memory can be owned by the user application by setting the parameter `marker_mode` to `ADQ_MARKER_MODE_USER_ADDR` and supplying the marker addresses in `marker_buffer_bus_address`. The marker buffers must be available for direct memory access (DMA). When using user supplied marker buffers, the user application is responsible for detecting updated marker values. Each marker consists of a 32-bit value starting at zero. The first time a buffer is available, the value 1 is written to the corresponding marker buffer. Each time new data is available in the buffer the marker value will increment by 1.

Note

A source code example demonstrating manual marker handling while transferring data to an AMD GPU is available for Windows platforms as `data_transfer_gpu`. See Section 15.2 for details on how to locate this software example.

10.3 Data Format

Correctly interpreting the record data requires knowledge about the *data format*. The various formats are enumerated using integer values and the format of a specific record propagates in the `header` field

`data_format`. The possible values are:

`ADQ_DATA_FORMAT_INT16`
: The record data consists of 16-bit signed integers and should be traversed using a pointer of matching type. The standard firmware represents samples using this format.

`ADQ_DATA_FORMAT_INT32`
: The record data consists of 32-bit signed integers and should be traversed using a pointer of matching type. The FWATD firmware (Section 5.6) represents samples using this format.

Important

Currently, the `data_format` is only set for records transferred to the host computer.

10.4 Data Transfer

Important

It is recommended to begin developing the user application by using the data readout interface described in Section 10.5 unless the use case involves advanced requirements.

The data transfer interface offers a low overhead, highly configurable method for data transfer at the expense of more complexity in the user application code. The interface is designed to allow the user to access the transferred data at the transfer buffer level and is typically used in applications where the final data destination is not the host computer's RAM, e.g. transferring data to a GPU.

10.4.1 Interface

The data transfer interface consists of four main functions:

- `StartDataAcquisition()` and `StopDataAcquisition()` starts and stops the data acquisition and data transfer processes;
- `WaitForP2pBuffers()` and `UnlockP2pBuffers()` allows access to the transfer buffers.

Refer to Sections A.4.4 and A.4.5 for detailed descriptions of these functions.

10.4.2 Program Flowchart

The expected program logic for a user application reading data directly via the data transfer interface is presented in Fig. 46. The steps are labeled on the left-hand side and are explained in the following list. Step 1a outlines the configuration for a typical transfer to the host computer's RAM. Step 1b describes the configuration for transferring data to a GPU or other peer-to-peer compatible devices.

Note

The source code example `data_transfer_gpu` demonstrates the program flow for the configuration in

- step 1a for Windows platforms; and
- step 1b for Linux platforms.

Refer to the example's `README` file for information on how to configure the different program flows. See Section 15.2 for details on how to locate this software example.

1. (a) The starting point for the flow diagram is a configured digitizer. Its parameters are expected to have been given values that reflect how the digitizer should behave during the acquisition. The general configuration process is outlined in Section 15.5. This section lists the parameter values that activate the data transfer process for a typical data transfer to the host computer's RAM. How to set up other parts of the digitizer is documented throughout the rest of this user guide.
   - Set `marker_mode` to `ADQ_MARKER_MODE_HOST_MANUAL`.
   - Set `write_lock_enabled` to 1 (default).
   - Set `packed_buffers_enabled` to 0 (default).
   - Set `transfer_records_to_host_enabled` to 1 (default).
   - For each *active* channel:
     - Set `nof_buffers` to a value in the range [2, `ADQ_MAX_NOF_BUFFERS`].
     - Decide the number of records per buffer (positive integer). Let this value be `N`.
     - Set `record_size` to the size of a record (in bytes).
     - Set `infinite_record_length_enabled` to 0 (default).
     - Set `record_buffer_size` to `N` times the record size.
     - Set `metadata_enabled` to 0.
   - For each *inactive* channel:
     - Set `nof_buffers` to 0 (default).

   (b) The starting point for the flow diagram is a configured digitizer. Its parameters are expected to have been given values that reflect how the digitizer should behave during the acquisition. The general configuration process is outlined in Section 15.5. This section lists the parameter values that activates the data transfer process for a typical data transfer to GPU or other peer-to-peer compatible devices. How to set up other parts of the digitizer is documented throughout the rest of this user guide.
   - Set `marker_mode` to `ADQ_MARKER_MODE_HOST_MANUAL`.
   - Set `write_lock_enabled` to 1 (default).
   - Set `packed_buffers_enabled` to 0 (default).
   - Set `transfer_records_to_host_enabled` to 0.
   - For each *active* channel:
     - Set `nof_buffers` to a value in the range [2, `ADQ_MAX_NOF_BUFFERS`].

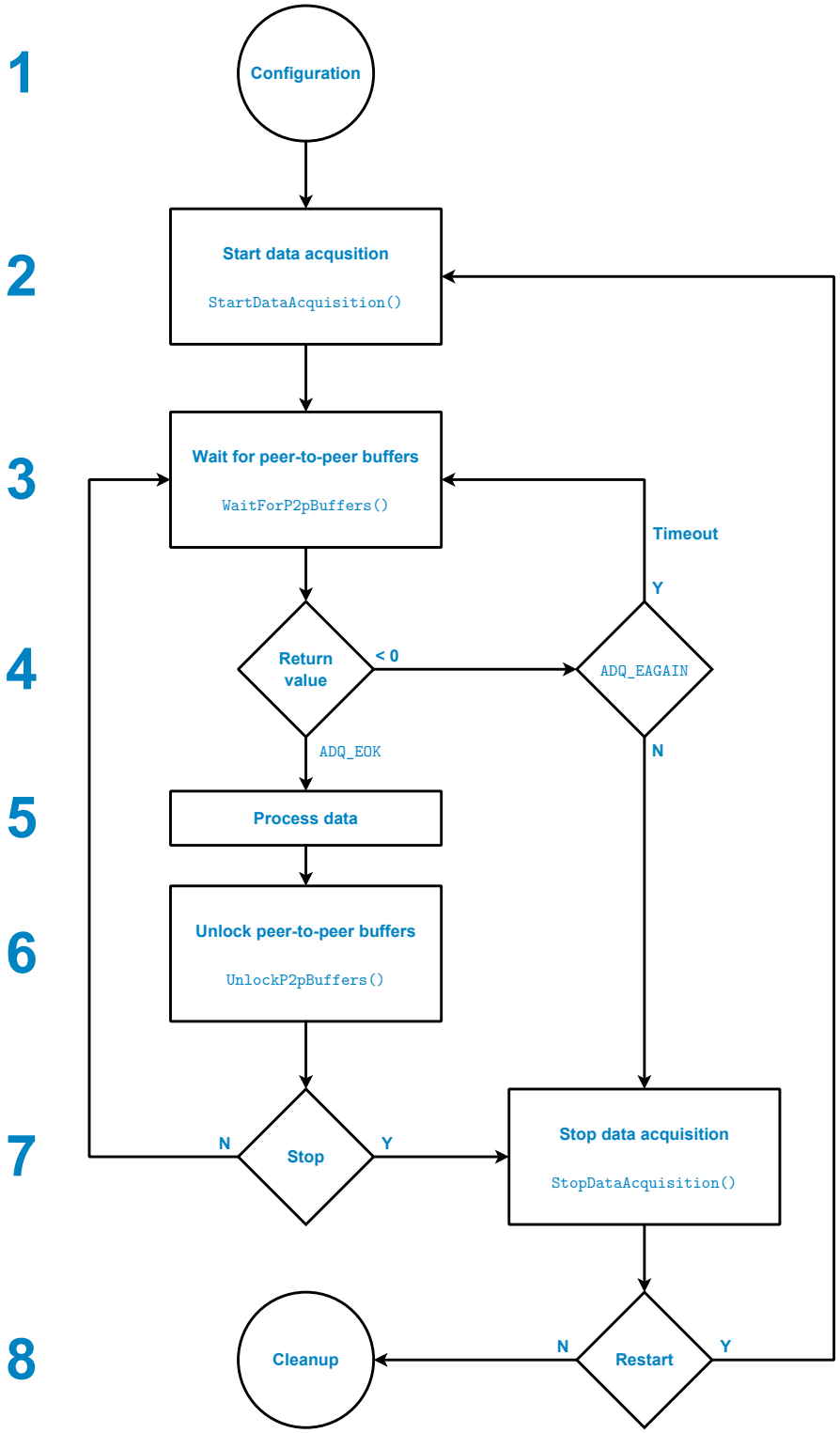


Figure 46: A flowchart for a user application interfacing directly with the data transfer process. The steps are labeled on the left-hand side and have a matching entry in Section 10.4.2.

  - Decide the number of records per buffer (positive integer). Let this value be `N`.
  - Set `record_size` to the expected size (in bytes) of a record.
  - Set `infinite_record_length_enabled` to 0 (default).
  - Set `record_buffer_size` to `N` times the record size.
  - Allocate `nof_buffers` of size `record_buffer_size` or bigger in the GPU (or other peer-to-peer compatible device).
  - For each allocated record buffer, enter its bus address into the `record_buffer_bus_address` array.
  - Set `metadata_enabled` to 0.
- For each *inactive* channel:
  - Set `nof_buffers` to 0 (default).

2. The data acquisition and data transfer processes are started simultaneously in a well-defined manner when `StartDataAcquisition()` is called. This effectively arms the digitizer which immediately begins observing the trigger conditions.

   Note

   The digitizer's parameters cannot be updated once the acquisition process is running.

3. The main program loop begins by waiting for the completion of at least one transfer buffer by calling `WaitForP2pBuffers()`. The parameter `timeout` is used to determine the behavior of the function call if data is not immediately available.

4. The function `WaitForP2pBuffers()` returns `ADQ_EOK` if at least one transfer buffer is available and negative values to indicate an error. Apart from the error code `ADQ_EAGAIN`, which indicates a timeout, and `ADQ_EINVAL` which indicates incorrect arguments, the negative values imply that an unrecoverable error has occurred and that the acquisition has been aborted. In this case, the user is expected to call `StopDataAcquisition()`.

5. The data processing step is the main purpose of a software application written for a digitizer. Whether it involves writing the data to disk to analyze at a later time, or performing real-time analysis, this user guide cannot offer information on implementation details since the requirements are highly application specific. However, a general guideline is not to perform computation-heavy operations in the loop (steps 3 to 7). This affects the balancing of the interface and can lead to overflows (Section 10.6). For high throughput applications, it is recommended to process and unlock the buffers of each channel in the same order as they appear in `completed_buffers` to avoid stalling the interface.

6. `UnlockP2pBuffers()` is called to make a buffer available to receive new data. Once a buffer has been unlocked, modification of its contents may happen at any time. Since the buffers of a channel are written in strict order, failing to unlock a buffer will cause the data transfer to halt once the transfer process reaches the locked buffer, even if other buffers are unlocked. If this condition persists, an overflow can occur. See Section 10.6 for more information.

   If `write_lock_enabled` is set to 0, buffers are overwritten as soon as new data is available, ignoring the locking mechanism with `UnlockP2pBuffers()`. Generally, this mode is *not* recommended

unless there are well-understood reasons why `UnlockP2pBuffers()` cannot be used and that real-time processing of buffers is guaranteed.

7. At the end of the main program loop, the application should determine if the acquisition should continue. If so, the program flow restarts from step 3. If the acquisition is complete or should stop for any other reason, the user is *required* to call `StopDataAcquisition()` to bring the data acquisition (and data transfer) process to a well-defined halt. The return value `ADQ_EINTERRUPTED` may be an expected error code if an acquisition is stopped prematurely.
8. Once the acquisition has been stopped, it is once again possible to modify the digitizer's parameters or to restart the acquisition with the same parameters by proceeding to step 2. If the application should exit, proceed with the cleanup phase outlined in Section 15.7.

10.4.3 Record Data Transfer Buffer Format

Records are stored as samples in the record transfer buffers without any padding between records. See Section 3 for details about the sample format.

10.4.4 Metadata Transfer Buffer Format

Metadata is stored in the format of an `ADQGen4RecordHeader` in the metadata transfer buffers. Certain fields in the header requires post processing to be valid, this processing is not available when the data transfer interface is used. See the documentation of the `ADQGen4RecordHeader` for details.

Important

Certain fields in the header requires post processing to be valid, this processing is disabled when the data transfer interface is used.

10.5 Data Readout

The data readout interface offers a low complexity, high flexibility method for making data available to the user application at the expense of a slight increase in computational overhead. Unless the use case involves advanced requirements on data throughput, it is recommended to begin developing the user application using this interface since the overhead is seldom a problem in practice.

The data readout interface outputs complete records and abstracts away the data transfer process and the transfer buffer handling. This reduces the complexity in the user application which can be written in a more straight-forward manner, processing one record at a time. Additionally, this interface is *thread-safe*, making it ideal to feed data to multi-threaded data processing applications. However, this interface is only supported when data is transferred to the host computer's RAM and requires that metadata is transferred from the digitizer (`metadata_enabled` set to 1). See Fig. 45 for an overview.

Important

The data readout interface is only supported when the transfer buffers are located in the host computer's RAM and requires that metadata is transferred from the digitizer.

10.5.1 Interface

The data readout interface consists of four main functions:

- `StartDataAcquisition()` and `StopDataAcquisition()` starts and stops the data acquisition, data transfer and data readout processes;
- `WaitForRecordBuffer()` and `ReturnRecordBuffer()` allows access to a target channel.

Refer to Sections A.4.4 and A.4.6 for detailed descriptions of these functions.

10.5.2 Record Buffers

Once there is at least one transfer buffer filled with data, its contents are translated into *record buffers* and passed on to the user application. These objects contains references to where the record `data` and its associated metadata, i.e. its `header` is located. The memory format of a record buffer is specified by the `ADQGen4Record` struct. The memory associated with a record buffer is owned by the API. Freeing these regions from the user application may lead to segmentation faults.

Important

The memory associated with a record buffer is owned by the API. Freeing these regions from the user application may lead to segmentation faults.

Note

By default, a record buffer always contains data from a full record. However, there is a mode which is useful in some situations, where `WaitForRecordBuffer()` can return partial record data. Refer to Section 10.5.7 for more information.

Dynamic Allocation

If the record length is dynamic (Section 9.1), or the digitizer is configured to continue on overflow (Section 10.6.3), the data from one record may end up split over several transfer buffers. In the interest of providing a low friction user experience by default, these events are hidden by copying data from the transfer buffers to dynamically allocated memory on the heap; so that the assumption of a contiguous `data` region hold true.

In a well-configured system, record data being copied from the transfer buffer only happens for a minority of the record buffers received by the user application. In the majority of cases, a copy is not required when the received record buffer can point directly to the transfer buffer memory (containing the data of a complete record). Ultimately, the frequency with which this mechanism is triggered will depend on the size of a record relative to the `record_buffer_size`.

The copy behavior is disabled when `incomplete_records_enabled` is set to 1, offloading the task of stitching together incomplete data to form the complete record to the user application. Refer to Section 10.5.7 for additional details.

Record buffers are allocated and resized as needed to handle the incoming data rate. However, the memory consumption of this process is constrained for each channel by the parameters `record_buffer_size_max`, `record_buffer_size_increment` and `nof_record_buffers_max`. Together they specify how large a record buffer is allowed to grow, the size added in each reallocation and the maximum number of buffers that may be allocated. By default, the process is unconstrained both in terms of the number of buffers and their maximum size. The record buffer memory is freed when `StopDataAcquisition()` is called.

10.5.3 Program Flowchart

The expected program flow for a user application reading data via the data readout interface is presented in Fig. 47. The steps are labeled on the left-hand side and are explained in the following list.

Note

The program flow in Fig. 47 is implemented in the software example `data_readout`. This example code in C is available as part of the release archive. See Section 15.2 for details on how to locate this example.

1. The starting point for the flow diagram is a configured digitizer. Its parameters are expected to have been given values that reflect how the digitizer should to behave during the acquisition. The general configuration process is outlined in Section 15.5. This section lists the parameter values that activates the data readout process. How to set up other parts of the digitizer is documented throughout the rest of this user guide.

    - Set `marker_mode` to `ADQ_MARKER_MODE_HOST_AUTO` (default).
    - Set `write_lock_enabled` to 1 (default).
    - Set `transfer_records_to_host_enabled` to 1 (default).
    - For each *active* channel:
        - Set `nof_buffers` to a value in the range [2, `ADQ_MAX_NOF_BUFFERS`].

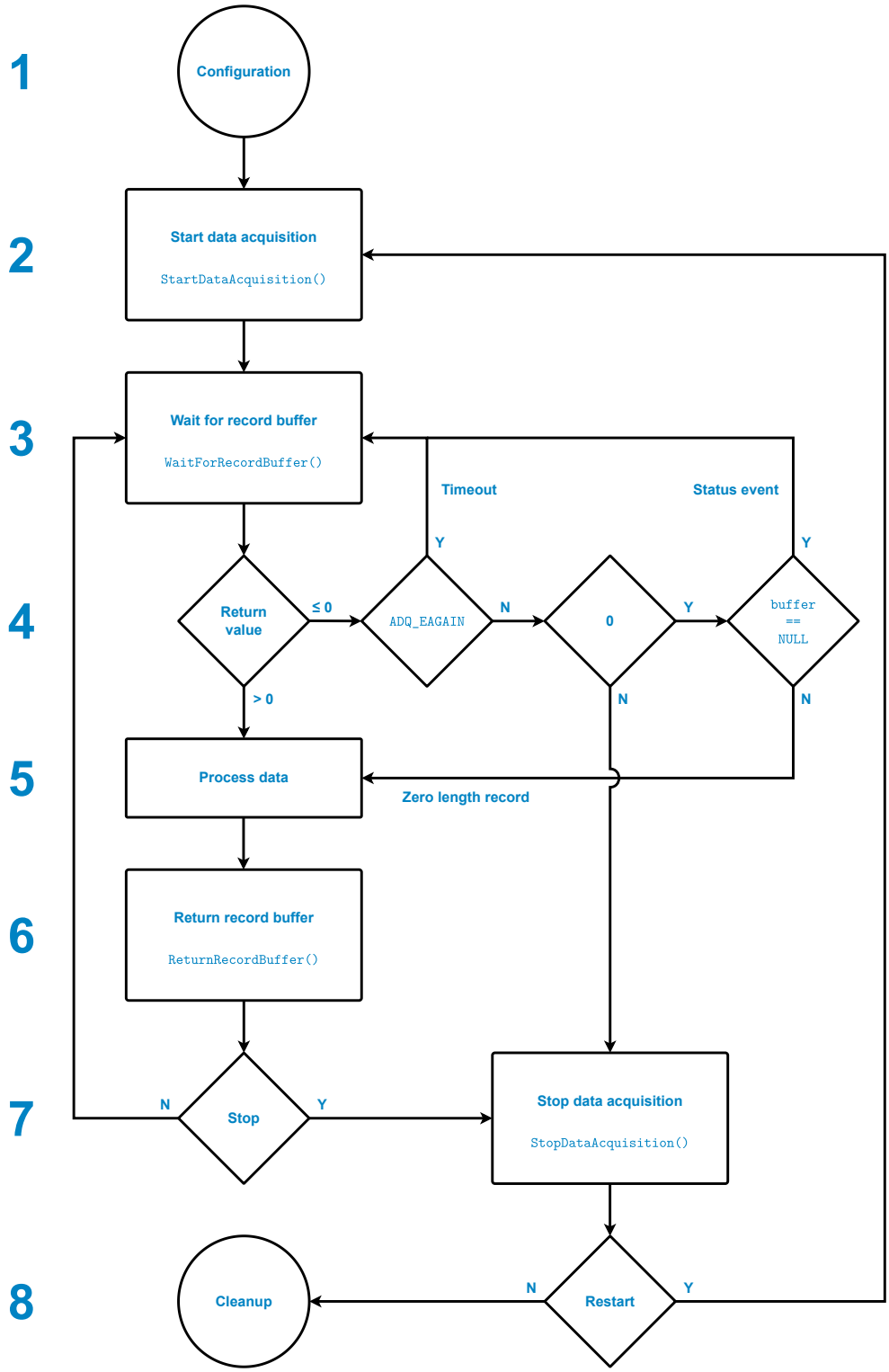


Figure 47: A flowchart for the data readout process. The steps are labeled on the left-hand side and have a matching entry in Section 10.5.3.

  - Set `metadata_enabled` to 1 (default).
  - Set `infinite_record_length_enabled` to 0 (default).
  - If the acquisition is configured for records with static length:
    * Set `record_size` to the expected size (in bytes) of a record.
    * Decide the number of records per transfer buffer (positive integer). Let this value be `N`.
    * Set `record_buffer_size` to `N` times the record size.
    * Set `metadata_buffer_size` to `N` times the size of a record header (`ADQGen4Record-Header`).
  - If the acquisition is configured for records with dynamic length or `continue_on_overflow_enabled` is set:
    * Set `record_size` to 0.
    * Set `record_buffer_size` to a suitable multiple of `record_buffer_size_step`.
    * Set `metadata_buffer_size` to a suitable multiple of the size of a record header (`ADQGen4RecordHeader`).

    Note

    The paired record buffer and metadata buffer are both ejected as soon as one of them reaches capacity. Consider the overhead caused by the metadata buffer always filling up first. Metadata buffers are comparatively cheap in terms of memory.

- For each *inactive* channel:
  - Set `nof_buffers` to 0 (default).

2. The data acquisition, data transfer and data readout processes are started simultaneously in a well-defined manner when `StartDataAcquisition()` is called. If this call is successful, a thread (Fig. 45) is created and the API *assumes control* of the digitizer. From this point, the user *must not* call any API functions other than those marked "🔀 Thread-safe" in Appendix A until the API releases the digitizer. Control is returned to the user if an error occurs or when `StopDataAcquisition()` is called.

   Note

   The digitizer's parameters cannot be updated once the acquisition process is running.

3. The data readout loop begins by waiting for a record buffer by calling `WaitForRecordBuffer()`. This operation may target a specific channel or use the special value `ADQ_ANY_CHANNEL` to return as soon as data is available on any of the active channels. The parameter `timeout` is used to determine the behavior of the function call if data is not immediately available.

4. The function `WaitForRecordBuffer()` returns negative values to indicate an error and positive values to indicate the number of bytes available in the record buffer's `data` region. Apart from the error code `ADQ_EAGAIN` which indicates a timeout, the negative values imply that an unrecoverable error has occurred and that the acquisition has been aborted. In this case, the user is expected to call `StopDataAcquisition()`. If the return value is zero, the function returned either a zero length record (Sections 9.5.1, 10.5.5) or a status event (Section 10.5.4). Use the value of `buffer` to differentiate between the two cases.

5. The data processing step is the main purpose of a software application written for a digitizer. Whether it involves writing the data to disk to analyze at a later time, or performing real-time analysis, this user guide cannot offer information on implementation details since the requirements are highly application specific. However, a general guideline is not to perform computation-heavy operations in the loop (steps 3 to 7). This affects the balancing of the interface and can lead to overflows (Section 10.6).

6. `ReturnRecordBuffer()` is called to make a record buffer available to receive new data. Once a reference to a record buffer has been registered with the API, modification of its contents may happen at any time. If the interface is consuming record buffers faster than the user can return them, the channel is said to be *starving*. If this condition persists, an overflow can occur. See Section 10.6 for more information.

7. At the end of the main program loop, the application should determine if the acquisition should continue. If so, the program flow restarts from step 3. If the acquisition is complete or should stop for any other reason, the user is *required* to call `StopDataAcquisition()` to bring the data acquisition (and data transfer) process to a well-defined halt. The return value `ADQ_EINTERRUPTED` may be an expected error code if an acquisition is stopped prematurely.

   Important

   When `StopDataAcquisition()` returns, any memory owned by the API is returned to the operating system. Attempting to access record buffers after this point may lead to access violations.

8. Once the acquisition has been stopped, it is once again possible to modify the digitizer's parameters or to restart the acquisition with the same parameters by proceeding to step 2. If the application should exit, proceed with the cleanup phase outlined in Section 15.7.

10.5.4 Status Events

Sometimes there is a need for the API to notify the user about certain events without propagating record data, e.g. signaling that data has been discarded due to an overflow (see Section 10.6). *Status events* are used for this purpose and are passed to the user application in the same way as record buffers—via `WaitForRecordBuffer()`. They may be recognized by the function returning the value 0 and the `buffer` being set to `NULL`. In such a situation, only the parameter `status` is valid and holds the event information. Receiving a status event from `WaitForRecordBuffer()` should *not* be matched by a symmetric call to `ReturnRecordBuffer()` since there is nothing to return.

10.5.5 Zero Length Records

Section 9.5.1 describes how the data acquisition process can be configured to emit records with no data as a form of real-time status event.

Receiving a zero length record from a call to `WaitForRecordBuffer()` will be characterized by the return value 0, the same as for status events. However, in the case of zero length records, `buffer` will point to valid memory and the `status flags` will signal `ADQ_DATA_READOUT_STATUS_FLAGS_OK`.

Important

Zero length records and status events can be separated by examining the value of the `status flags` (`ADQ_DATA_READOUT_STATUS_FLAGS_OK` for zero length records), or the value of `buffer` (`NULL` for status events).

10.5.6 Discarded Records

If a record cannot be transferred in its entirety to the user application, it is discarded with the rationale that the user would have done the same upon realizing this fact. In its place, a status event (Section 10.5.4) is emitted signaling `ADQ_DATA_READOUT_STATUS_FLAGS_DISCARDED` in the `status flags` of `WaitForRecordBuffer()`.

Records are discarded as the result of an overflow caused by a data rate imbalance in the device-to-host interface (Section 10.6), or by a dynamically allocated record buffer (Section 10.5.2) reaching its maximum capacity, as defined by `record_buffer_size_max`.

If this behavior is undesired, the data readout interface can be configured to emit partial record data in the form of *incomplete records*. See Section 10.5.7 for additional details.

10.5.7 Incomplete Records

The default behavior of the data readout interface is to return complete records, i.e. one record buffer (`ADQGen4Record`) holds exactly the data associated with one acquired record. This is to maintain low friction between the mental model of the data acquisition process and the user application. However, there are situations where this behavior becomes undesired or even impractical. For example, there are obvious issues with memory allocation if the record length is infinite (`record_length` is set to `ADQ_INFINITE_RECORD_LENGTH`). Another example is if the copy mechanism described in Section 10.5.2 is suspected to limit performance or should be disabled for other reasons.

The parameter `incomplete_records_enabled` controls whether or not `WaitForRecordBuffer()` is allowed to return partial data on success and is zero (disabled) by default. Whether a record is incomplete or complete is communicated via a flag in the parameter `status` for each successful call to the function. Additionally, record headers will propagate to the user application together with the record buffer that contains the *last* partial data. In the case of a record with infinite length, no metadata is transferred.

Note

The mechanism to propagate partial record data increases the flexibility of the data readout interface, at the cost of increased complexity in the logic of the user application. It is only recommended if required by the use case.

Note

In the case of a record with infinite length, no metadata is transferred.

Modifications to the Program Flowchart

The expected program flow for a user application reading incomplete record data via the data readout interface is essentially identical to the steps described in Section 10.5.3. The only differences are in

steps 1 and 5: configuration and data processing. The entries in the list below are intended to supersede the corresponding entries in the list in Section 10.5.3. Step 1a describes the configuration for records with finite length and step 1b describes the configuration for records with infinite length.

1. (a) Start from the configuration outlined in step 1 (Section 10.5.3) for records with dynamic length and make the following adjustments:
   - Set `incomplete_records_enabled` to 1.
   - Set `nof_record_buffers_max` to 0.

   (b) Start from the configuration outlined in step 1 (Section 10.5.3) for records with dynamic length and make the following adjustments:
   - Set `infinite_record_length_enabled` to 1.
   - Set `metadata_enabled` to 0.
   - Set `metadata_buffer_size` to 0.
   - Set `incomplete_records_enabled` to 1.
   - Set `nof_record_buffers_max` to 0.

5. The data processing step increases in complexity since the user application also needs logic to decide how to handle partial data, in addition to the logic performing the use case specific processing. As in the case of complete records, this user guide cannot offer information on implementation details since the requirements varies with the use case. However, there are essentially two strategies:
   - process the partial data immediately; or
   - stitch together the complete record by copying the partial data until the last record buffer is emitted, then proceed with the processing step.

   The latter is only applicable for finite records (step 1b). Regardless of the chosen strategy, the user application should query the `status` parameter propagated next to the record buffer in the call to `WaitForRecordBuffer()` for information about the record buffer. The member `flags` is a bitmask that will signal `ADQ_DATA_READOUT_STATUS_FLAGS_INCOMPLETE` when the emitted record buffer contains partial data. The corresponding flag bit is set to zero for the last batch of partial data that completes the record. The return value of `WaitForRecordBuffer()` indicates the payload, i.e. the amount of partial `data` available for reading.

10.5.8 Optimizing Throughput

While the data readout interface offers an intuitive view of the data stream where one event corresponds to one record, this view has one weakness in practice: high event rates. Since the channels are made thread safe by the API, there is a certain overhead associated with processing and dispatching record buffers to the user application (Fig. 45). In practice, that means that each host system has a point where trigger rate starts to negatively affect the effective bandwidth. Precisely where this drop-off point is located depends on the host system's specification. Generally, shorter records allow higher trigger rates. Running at rates at or above 100 kHz can impact the performance of this interface in its default configuration.

To tackle this issue, the concept of *arrayed* record buffers is introduced. In this mode (which is disabled by default), each successful call to `WaitForRecordBuffer()` emits an array object holding one or several record buffers: `ADQGen4RecordArray`. Since multiple records buffers are emitted together, this reduces the activity of the thread safe data readout interface while still propagating the same amount of record data.

This mechanism is controlled by the data readout parameter `nof_record_buffers_in_array`. When the parameter is set to a positive value, arrays containing the specified number of records are emitted by `WaitForRecordBuffer()`. The only allowed negative value is the special value `ADQ_FOLLOW_RECORD_TRANSFER_BUFFER` which indicates that the contents of the emitted array should exactly represent the contents of a record transfer buffer. Setting the parameter to zero (the default value) disables the array mechanism, causing `WaitForRecordBuffer()` to emit `ADQGen4Record` objects.

Note

Shorter records allow higher trigger rates. User applications running at rates at or above 100 kHz may need to utilize arrayed record buffers to stay performant in these situations.

10.6 Overflow

Important

By default, the data acquisition stops in the event of an overflow *unless*

- the user has activated the *continue on overflow* mechanism, described in Section 10.6.3; or
- the digitizer is running the FWATD firmware, in which case the overflow behavior differs significantly from how it is described in this section. See Section 5.6.6 for more information.

An *overflow* in this context means the result of a data rate imbalance where data is forced to be discarded. There are two possible causes, either

1. the data rate of the acquisition process exceeds the bandwidth of the physical interface; or
2. the transfer buffers are not being made available for new data at a sufficient rate.

Sections 10.6.1 and 10.6.2 describes the two cases in more detail. Section 10.6.3 describes how the digitizer can be made to *continue* its data acquisition process in the event of an overflow and what that means for the data transfer process and the user application. Refer to `GetStatus()` for information on how to query the digitizer for the overflow status.

10.6.1 Physical Interface (case 1)

ADQ3 series digitizers are equipped with on-board memory whose purpose is to act as a buffer for the physical interface. This buffer is required since the physical interface may experience temporary stalls at any time, leaving the digitizer with two options: discard the data, or store and transfer it at a later time. The latter option is chosen as long as the memory is not filled to capacity, but if the imbalance continues for an extended period of time, discarding data will be the only option. If an overflow occurs, the acquisition process will either come to a halt or continue to the best of its ability (see Section 10.6.3).

In both cases, the contents of the on-board memory at the time of the overflow remain intact and can be transferred safely. Refer to `GetStatus()` for information on how to query the digitizer for the overflow status.

10.6.2 Transfer Interface (case 2)

The other critical point is when the transferred data is propagated to the user application via one of the two available interfaces:

1. either `WaitForRecordBuffer()` and `ReturnRecordBuffer()`; or
2. `WaitForP2pBuffers()` and `UnlockP2pBuffers()`.

Regardless of interface, the user is expected to perform symmetrical operations. For every transfer buffer given to the user application via `WaitForP2pBuffers()`, a corresponding call to `UnlockP2pBuffers()` to unlock it is expected. For every record buffer propagated via `WaitForRecordBuffer()`, a corresponding call to `ReturnRecordBuffer()` returning the record buffer is expected. Note that a call to `WaitForRecordBuffer()` is not *guaranteed* to emit a record buffer. Errors and status events are the exception to this rule. See step 4 in the program flow chart in Fig. 47 Section 10.5.3 for additional details.

If the user application fails return memory to the API at a sufficient rate, the digitizer will have nowhere to transfer new data. This stops the data transfer process and causes the digitizer's on-board memory to start filling up and potentially trigger the overflow behavior described in Section 10.6.1. When this happens, the interface is said to be *starving*. It is worth repeating that no data is lost until the digitizer's on-board buffer memory overflows. Data waiting to be transferred remains intact when the data transfer process suspends its operations.

To avoid starving the interface, the user first needs to ensure that any processing step (Section 10.4.2, step 5 and Section 10.5.3, step 5) is able to handle the sustained acquisition data rate. For example, acquiring data at an average rate of 2 GB/s and writing this to a solid-state drive (SSD) with an average write speed of 500 MB/s is not sustainable. Second, assuming the processing step is capable of handling the target data rate, the user will need to balance the transfer buffer sizes. Each use case will have its own optimal transfer buffer size, meaning this parameter must be tuned by the user. In general, the trade-off is between latency (smaller buffers) and throughput (larger buffers).

Note

To avoid starving the interface, the user needs to make sure that the processing step (Section 10.4.2, step 5 and Section 10.5.3, step 5) is able to handle the acquisition data rate.

10.6.3 Continue on Overflow

By default, the data acquisition halts when an overflow is detected. This behavior can be changed via the parameter `continue_on_overflow_enabled`. When activated, the digitizer will continue to acquire and transfer records in the event of an overflow. However, this behavior will result in some records being abruptly cut short, rendering them incomplete with no way of recovering the data that was lost. This effectively creates a situation where the length of a record is *dynamic* but determined by events outside the user's control, as opposed to events defined by the data acquisition process (Section 9.1). Since the resulting data flow behaves in the same way for both cases, the data transfer process must

be configured to support transferring records with dynamic length if `continue_on_overflow_enabled` is set. The necessary configuration is outlined in step 1 in Section 10.5.3.

By default, records with incomplete data are discarded by the data readout interface (Section 10.5.6). To change this behavior, incomplete records must be allowed to propagate through the interface. Refer to Section 10.5.7 for more information. A record where data definitely has been lost will signal `ADQ_RECORD_STATUS_OVERFLOW` in the `header` field `record_status`.

Note

The digitizer cannot be configured to continue on overflow if the `record_length` is set to `ADQ_INFINITE_RECORD_LENGTH`.

Hysteresis

If the acquisition rate is such that the on-board memory exists in a constant state of overflow, and `continue_on_overflow_enabled` is set, the `overflow_hysteresis` may need adjustment to ensure that some records still propagate in their entirety.The hysteresis is given as a percentage of the `dram_size` and represents an *offset* from the maximum capacity. Following an overflow, no new data is written to the on-board memory until it has been emptied by *at least* this amount. If at least one complete record fits within this margin, the user application will eventually receive a complete record—regardless of any perpetual overflow condition.

Example

Consider a `dram_size` of 8 GiB and an `overflow_hysteresis` of 3%. Following an overflow, no new data will be written to the on-board memory until

$$\texttt{dram\_size} \cdot \texttt{overflow\_hysteresis} = 8 \cdot 1024^3 \cdot 0.03 = 245.76 \text{ MiB}$$

has been made available.

The `overflow_hysteresis` is subjected to rounding and not intended to be controlled with high precision. The value read in a call to `GetParameters()` may be different from the requested hysteresis but reflects the one used by the digitizer.

10.7 Eject

A partially filled transfer buffer (Section 10.1) can be made available to the user application by *ejecting* it. Normally, a transfer buffer becomes available to the user when filled to capacity. Transfer buffers may be ejected in a variety of ways, namely:

- by a software request,
- when a timeout is reached,
- at the end of a record; or
- when the signal output from one of the pattern generators (Section 7.1) is logic high.

The eject mechanism is controlled by the data transfer parameters `eject_buffer_source` and `eject_buffer_timeout`.

Note

The transfer buffer will only be ejected if it is partially filled with data. Ejecting an empty transfer buffer has no effect.

Important

Frivolously ejecting the transfer buffers may reduce the maximum throughput.

Software request

A partially filled transfer buffer is ejected via a software request by calling `EjectTransferBuffer()`. The timing of the function call relative to data or external input is not guaranteed. This function is only intended to be used in specific cases, generally to eject the last buffer of an acquisition. This eject source is always enabled, regardless of the value of `eject_buffer_source`.

Timeout

The timeout is enabled by setting the `eject_buffer_timeout` to a positive value. A partially filled transfer buffer will be ejected when no data has been written for the specified amount of time. The timeout is cleared by each record, meaning that there is a trigger frequency for which eject events of this type will not be triggered. This mode of operation should not be confused with ejecting transfer buffers periodically. If this is desired, use one of the pattern generators to create a periodic signal and set `eject_buffer_source` to target one of them, e.g. `ADQ_FUNCTION_PATTERN_GENERATOR0`.

Record stop

A partially filled transfer buffer is ejected after each record. For records with static length, this should normally not be used since the same effect can be achieved by setting the `record_buffer_size` to match the record length.

Pattern generator

The pattern generator may be used to generate an arbitrary signal from external input or timers to eject the buffer. A partially filled transfer buffer is ejected when the pattern `output_value` is logic high.

10.8 Calculating the Data Rate

To calculate the effective data rate of *one channel*, use the relation presented in (29). The calculation assumes no metadata, a trigger rate of $f_{trigger}$ and that the trigger period is greater than the record length, i.e. that records do not overlap.

$$\texttt{bytes\_per\_sample} \cdot \texttt{record\_length} \cdot f_{trigger} \quad [\text{ bytes / s }] \tag{29}$$

If metadata is enabled (`metadata_enabled` is a nonzero value), each trigger also adds the transfer of an `ADQGen4RecordHeader`. Thus, (29) becomes

$$(\texttt{bytes\_per\_sample} \cdot \texttt{record\_length} + \texttt{sizeof}(\texttt{ADQGen4RecordHeader})) \cdot f_{trigger} \quad [\text{ bytes / s }] \quad (30)$$

To calculate the total data rate, sum the contributions from each active channel.

Example

An ADQ32 with a base sampling rate of 2500 MSPS is configured to acquire data on both channel A and channel B. Channel A is set up to acquire a record with 2000 samples every rising edge detected on the TRIG port. The periodic signal input on the TRIG port has a frequency of 100 kHz. Metadata is active. Using (30), the data rate can be calculated as

$$(2 \cdot 2000 + 64) \cdot 100 \cdot 10^3 = 406400 \cdot 10^3 = 406.4 \text{ MB/s}.$$

Channel B is set up to acquire a record with 100000 samples every rising edge detected on the SYNC port. The periodic signal input on the SYNC port has a frequency of 2 kHz. Metadata is not active. Using (29), the data rate can be calculated as

$$2 \cdot 100000 \cdot 2 \cdot 10^3 = 400000 \cdot 10^3 = 400.0 \text{ MB/s}.$$

Thus, the total sustained data rate is

$$406.4 \cdot 10^6 + 400.0 \cdot 10^6 = 806.4 \text{ MB/s}.$$

Some firmware types are specifically aimed at reducing the amount of data transferred over the device-to-host interface without sacrificing acquisition rate. For example, the FWATD firmware (Section 5.6) features hardware accelerated accumulation of records. This allows higher acquisition rates than what can be supported by the physical interface by virtue of the accumulator significantly reducing the amount of data transferred between the digitizer and the endpoint. Such aspects affect the calculations presented in (29) and (30) which then becomes worst-case values.

11 Test Pattern

The digitizer has a built-in test pattern generator which replaces the digitized data from the ADC with a digitally generated sequence of values. This can be useful for debugging purposes. The test pattern is disabled by default, but can activated on a per-channel basis by specifying an appropriate value for the test pattern `source` parameter. The following sources are available:

`ADQ_TEST_PATTERN_SOURCE_COUNT_UP`

A sawtooth pattern which counts upwards from −32768 to 32767, with each sample incrementing in value by 1.

`ADQ_TEST_PATTERN_SOURCE_COUNT_DOWN`

A sawtooth pattern which counts downwards from 32767 to −32768, with each sample decrementing in value by 1.

`ADQ_TEST_PATTERN_SOURCE_TRIANGLE`

A triangle pattern which first counts upwards from −32768 to 32767, followed by a downward count from 32767 to −32768. At the boundary between counting upwards and downwards, the sample value (32767 and −32768) will be repeated twice.

Note

The test pattern is not affected by the digital gain and offset signal processing step described in Section 5.1.

12 System Manager

The system manager is a dedicated hardware component responsible for the digitizer's firmware management, temperature monitoring, supply voltage monitoring, overtemperature protection and fan control (where applicable). The system manager has two communication interfaces:

- an interface integrated into the digitizer's device-to-host interface, e.g. the PCIe interface.
- a dedicated USB interface located at the edge of the digitizer. The interface connector is *not* accessible via the front panel and its format differs between digitizer models:
  - ADQ30-PCIe, ADQ32-PCIe and ADQ33-PCIe are fitted with either an USB-C connector or a micro USB connector, depending on the date of production.
  - ADQ36-PXIe is fitted with a micro USB connector.

The dedicated USB interface acts as the fallback interface if the digitizer should ever be put in a state where the main device-to-host interface is unavailable. While rare, this can happen for various reasons, as described in the following sections.

12.1 Firmware

The digitizer's firmware memory can store several firmware *images*. To manage these, the software tools *ADQUpdater* or *ADQAssist* are used. Refer to the ADQUpdater user guide [5] for more information.

If the digitizer firmware has somehow been compromised and communication via the device-to-host interface is not possible, a fallback option is provided via the dedicated USB connector (described in Section 12).

12.1.1 Channel Configuration

Some ADQ3 series digitizers support several channel configurations on the same hardware. This affects the *number of channels* and their *base sampling rate*. For example, ADQ32 can either run as a two-channel digitizer with a base sampling rate of 2.5 GSPS, or as a one-channel digitizer with a base sampling rate of 5 GSPS. This is controlled via the active firmware image. Below is a typical output from listing the available firmware images in ADQUpdater. Two images are listed: a two-channel 2.5 GSPS version as image 0, and a one-channel 5 GSPS version as image 1.

```
Image  0 (default)  (current)
--------
         Size: 13 MB (13585656 B)
     Revision: 2023.2
  Description: 2CH-FWDAQ-PCIE
  Part number: 400-023-000
          MD5: 1A7BB25D794534072B72A32262D1F57E
      Address: 0x02000000
    Timestamp: 2023-06-30T07:55:04Z

Image  1
--------
         Size: 13 MB (13429412 B)
     Revision: 2023.2
  Description: 1CH-FWDAQ-PCIE
  Part number: 400-023-001
          MD5: FF319E56988132CF9BDF55FCC803EACA
      Address: 0x01000000
    Timestamp: 2023-06-30T07:59:18Z
```

Changing the channel configuration involves specifying the desired firmware image as the new *default image* and then power cycling the device. In general, this also includes the host system since the device-to-host interface has to be renegotiated.

Important

Changing the channel configuration requires that the device is power cycled. In general, this also includes the host system since the device-to-host interface has to be renegotiated.

12.2 License Management

Certain features of the digitizer are locked behind *licenses*. These are stored in the on-board nonvolatile memory. By default, the FWDAQ firmware license is enabled on all digitizers. Other firmware licenses, such as the one required to run the FWATD firmware, constitute an add-on purchase for each digitizer and are not enabled by default.

If an add-on firmware license was purchased at the same time as the digitizer hardware, the license will already be enabled on the unit when it is shipped and no action needs to be taken by the user. If the firmware license is instead purchased for a unit in the field, a field update of the license must be performed. This is done using the *ADQLicenseUtil* software tool.

To start, the unit-specific *DNA* identifier should be read out:

```
> adqlicenseutil --read-dna

Unit 1 (ADQ32) - Serial: SPD-01234, DNA: 00000000400200012345678 9ABCDEF000
```

The output from this command can then be sent to your sales representative and a license file with the new license added will be generated. This file will be unique to the digitizer from which the DNA identifier was read, and will not work for other digitizers. To write the new license file to the unit, execute the following command:

```
> adqlicenseutil --write-license SPD-01234_FWDAQ_FWATD_14-Dec-2022.lic
```

The user can query the status of the digitizer's license situation, with respect to the active firmware, by calling `GetStatus()` with `ADQ_STATUS_ID_LICENSE` as the target identifier.

12.3 Temperature Monitoring

The system manager continually monitors the operating temperatures of components in the digitizer hardware such as the FPGA, ADCs and voltage converters. Each temperature sensor has a set limit, which once exceeded, will cause the system manager to trigger the overtemperature protection mechanism. See Section 12.3.1 for more information.

The user can query the API for the current values of the digitizer's temperature sensors by calling `GetStatus()` with `ADQ_STATUS_ID_TEMPERATURE` as the target identifier. The values are organized into an array of `sensor` entries in `ADQTemperatureStatus`.

12.3.1 Overtemperature Protection

The overtemperature protection mechanism can trigger two types of faults:

1. a *recoverable* overtemperature fault; and
2. an *unrecoverable* overtemperature fault.

Both faults are signaled via the STAT LED (see Section 13.1), where the unrecoverable overtemperature fault takes priority if both are asserted. It is also possible to remotely list the current status of the digitizer via ADQUpdater.

Recoverable Overtemperature Fault

If the recoverable overtemperature fault is triggered, the system manager attempts to reduce the temperature by reducing the dynamic power consumption. The ADCs are switched off and the clock system is disabled, immediately halting the flow of data. In this state, it is still possible to communicate with the digitizer over the device-to-host interface. However, any ongoing acquisition will be aborted and the digitizer will not be able to be reinitialized until the fault is cleared. Recovering from this fault requires manual intervention, signaling that the fault is acknowledged by the user and that an attempt should be made to power up the ADCs and the clock system. The fault is cleared by issuing a *soft reset* of the system manager via ADQUpdater. If the fault is cleared but the factors causing the high operating temperature remain, the system manager will trigger the fault once again.

Unrecoverable Overtemperature Fault

Should the actions taken by the recoverable overtemperature fault not be sufficient to reduce the digitizer's operating temperature, the unrecoverable overtemperature fault will be triggered. In this state, core supply voltages are disabled to avoid damaging system components. This means that the device-to-host interface will forcefully close, which can result in a freeze of the host system depending on the interface activity at the time of the fault. Once this fault is triggered, the digitizer will not be able to operate

properly until it has been completely power cycled. Communicating with the system manager will only be possible via the dedicated USB port.

Important

Once the unrecoverable overtemperature fault is triggered, the device-to-host interface will forcefully close. This may result in a freeze of the host system depending on the interface activity at the time of the fault.

13 Front Panel LEDs

This section describes the function of the digitizer's front panel LEDs.

13.1 STAT

The LED labeled "STAT" is a multipurpose LED indicator controlled by the system manager (Section 12). It is used to signal various status information about the digitizer. During normal operation, the LED is constantly lit green, indicating that there are no issues. If the LED is not green, Table 10 can be used to determine the meaning. If multiple errors occur at the same time, the LED will signal the one with highest priority. Additionally, the user can call `Blink()` to initiate a five second 1 Hz blue blinking pattern. This is useful when operating multiple digitizers.

Table 10: The states of the STAT LED, sorted in descending priority.

LED state	Description
Blinking red, 5 Hz	The unrecoverable overtemperature fault has been triggered. See Section 12.3.1 for more information.
Blinking orange, 5 Hz	The recoverable overtemperature fault has been triggered. See Section 12.3.1 for more information.
Blinking red, alternating long/short	Waiting for power supplies to start up.
Constant purple	Communication with the digitizer's firmware memory cannot be established.
Blinking purple, 1 Hz	A sensor reading has failed.
Blinking yellow, 1 Hz	Failed to load the digitizer's firmware.
Alternating red/yellow, 2 Hz	No valid digitizer firmware found.
Constant green	Normal operation.

Note

ADQUpdater can be used to query the digitizer for the current status. Refer to the ADQUpdater user guide [5] for more information.

13.2 RDY

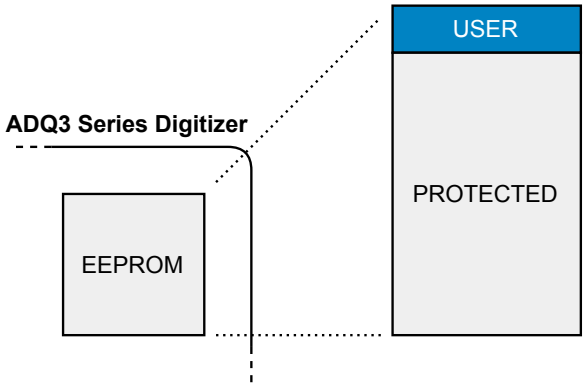
The LED labeled "RDY" is an activity indicator for the data acquisition process. When the LED is lit yellow, the data acquisition process (for at least one channel) is waiting for a trigger event to be observed. When a trigger event is detected, the LED is turned off briefly. The rate of the blinking is limited to 10 Hz, even if the trigger rate is higher.

13.3 USER

The LED labeled "USER" is controlled via the FPGA development kit. The default behavior for the LED is to always be off.

14 EEPROM

Each ADQ3 series digitizer is fitted with a nonvolatile memory to store calibration data as well as information used for device identification. This data is protected and cannot be modified by the user. However, there's a dedicated *user area* where the user is free to store data in a persistent manner. The size of the area is defined by the constant parameter `eeprom_user_area_size`.



The memory interface is defined by the two functions `WriteEeprom()` and `ReadEeprom()`. Refer to their respective documentation in Appendix A for more information.

15 API

This section describes the structure of the application programming interface (API) and the general principles of composing an application that interfaces with the digitizer. The API consists of two main components:

- a platform-specific *shared object library*: `ADQAPI.dll` on Windows, `libadq.so` on Linux; and
- a *header file*: `ADQAPI.h`.

The header file defines the constants and function signatures of the library using the C programming language, but it is possible to successfully interface with the digitizer from any language with a *foreign function interface* (FFI) compatible with C. The ADQAPI uses two types of objects (classes): the control unit and the device object. The control unit manages the connection between the digitizers and the host computer, and is responsible for creating the device object. The device object handles the communication with each device. The API functions are categorized into three main sets:

ADQAPI-specific functions
: Functions purely related to the API itself and not the operation of a digitizer. These functions do not require a reference to the control unit, e.g. `ADQAPI_ValidateVersion()`.

ADQ control unit functions
: Functions which interface with the control unit for tasks such as finding and identifying digitizers, e.g. `ADQControlUnit_ListDevices()`.

ADQ functions
: Functions which interface with a specific digitizer, e.g. `SetParameters()`.

The normal control flow for an application managing the digitizer consists of the following parts:

1. Identification (Section 15.3)
2. Initialization (Section 15.4)
3. Configuration (Section 15.5)
4. Acquisition (Section 15.6)
5. Cleanup (Section 15.7)

The flow between these parts are visualized in Fig. 48 and described in the following sections using the C programming language to provide context.

The initialization and configuration parts both use changes to the digitizer's *parameter space* when configuring the digitizer. See Section 15.8 for detailed information on interacting with the parameter space.

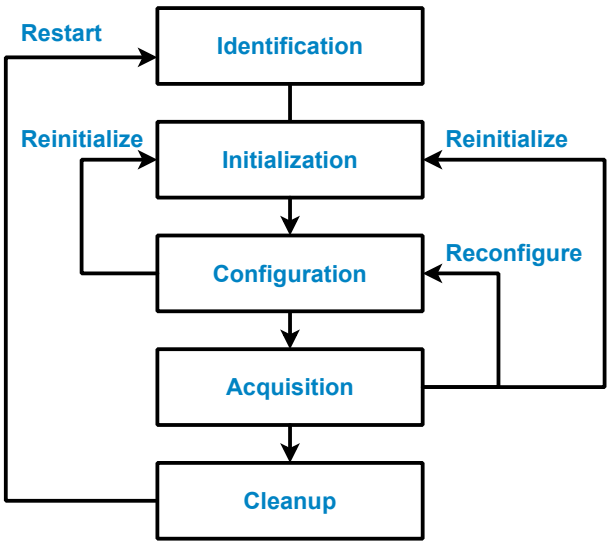


Figure 48: Typical control flow of an application interfacing with an ADQ3 series digitizer.

15.1 SDK Installation

The software development kit (SDK) contains the ADQAPI, drivers and other tools required to successfully interface with the digitizer. The installation procedure for Microsoft Windows and Linux is described in the following sections.

15.1.1 Installing the SDK (Windows)

For Microsoft Windows the SDK is installed by running

```
TSPD_SDK_windows_<version>.exe
```

and following the instructions. The `<version>` part of the file name is the version number.

15.1.2 Installing the SDK (Linux)

The SDK is supported for several Linux distributions and versions. The complete list can be found in the document listing operating system support [6]. The installation files are included in

```
TSPD_SDK_linux_<version>.tar.gz
```

where `<version>` is the version number. The archive contains installation files for the target distribution for the ADQAPI, drivers and other tools required to successfully interface with the digitizer. The `README` file, located in the root directory of the archive, describes the installation procedure in detail for the different distributions.

15.2 Software Examples

The software examples mentioned throughout this document are available on a model and firmware basis as part of the *release archives*. They are located in the subdirectory `examples/` and separated by programming language. Some digitizer features are only demonstrated in one programming language.

15.3 Identification

In the identification phase, the available digitizers are listed and selected for setup. The examples below uses the C API which can be used in both C and C++ applications. For Python, the program flow differs slightly, refer to Section 16 and the Python example. The digitizer identification is handled by the *ADQ control unit*. The first step is to create the control unit:

```
void *adq_cu = CreateADQControlUnit();
if (adq_cu == NULL)
  /* Handle error */
```

The variable `adq_cu` is a pointer to the control unit object and must be passed to all API calls interfacing with a digitizer. The error logging can now be enabled with

```
ADQControlUnit_EnableErrorTrace(adq_cu, LOG_LEVEL_INFO, ".");
```

Enabling the log file is not required but highly recommended since the API communicates the *cause* of errors via log messages. After the control unit has been created, the available devices can be enumerated:

```
struct ADQInfoListEntry *adq_list = NULL;
unsigned int nof_devices;
if (!ADQControlUnit_ListDevices(adq_cu, &adq_list, &nof_devices))
  /* Handle error */
```

If successful, the `nof_devices` variable will hold the number of digitizers connected. Assuming at least one digitizer is available (`nof_devices` > 0), the first digitizer can be set up:

```
int device_to_open = 0; /* Indexing starts at 0 */
if (!ADQControlUnit_SetupDevice(adq_cu, device_to_open))
  /* Handle error */
```

If no errors have occurred, the digitizer is now set up and ready to be used. For example, the current parameter values can be read with `GetParameters()`:

```
int adq_num = 1; /* Indexing starts at 1 */
struct ADQParameters adq_parameters = {0};
if (ADQ_GetParameters(adq_cu, adq_num, ADQ_PARAMETER_ID_TOP, &adq_parameters)
    != sizeof(adq_parameters))
  /* Handle error */
```

Note that the identification number (`adq_num`) used in device calls starts at 1.

15.4 Initialization

In the initialization phase, parameter changes that disrupt the digitizer's operation are performed. See Section 15.8 for details on interacting with the parameter space.

There are two examples of initialization phase parameters:

- `ADQClockSystemParameters` (Section 15.4.1)
- `ADQInputRoutingParameters` (Section 15.4.2)

15.4.1 Clock System

Changing the clock system configuration is disruptive since it resets the ADCs, the clock circuitry, and other parts of the digitizer's data path. See Section 4 for details on the available alternatives for configuring the clock system. By default, the clock system is set up to use the internal reference clock and clock generator, which means that this step can be safely skipped if this is the desired clock system configuration.

Note

Configuring the clock system parameters may be safely skipped if the digitizer should use the internal reference clock and the default base sampling rate.

The code snippet below sets up the clock system for an external reference clock via the CLK port:

```
/* Allocate a variable to hold the clock system parameters. */
struct ADQClockSystemParameters clock_system;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_CLOCK_SYSTEM,
                                      &clock_system);
if (result != sizeof(clock_system))
{
  /* Handle error */
}

/* Enable external reference on the CLK port, with low jitter mode enabled. */
clock_system.reference_source = ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK;
clock_system.reference_frequency = 10e6;
clock_system.low_jitter_mode_enabled = 1;

/* Set up the clock system. */
result = ADQ_SetParameters(adq_cu, adq_num, &clock_system)
if (result != sizeof(clock_system))
{
  /* Handle error */
}
```

The code snippet below shows a more advanced use case where the external reference clock frequency and the target sampling rate differ from the default values:

```
/* Enable external reference on the CLK port, modified reference frequency
   and sampling rate. */
clock_system.reference_source = ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK;
clock_system.reference_frequency = 25e6;
clock_system.sampling_frequency = 2475e6;
clock_system.low_jitter_mode_enabled = 0;
```

The code snippet below shows a use case where external clock generation is used, where the full 2250 MHz clock must be supplied via the CLK port:

```
/* Use external clock generation to sample at 2250 MSPS. */
clock_system.clock_generator = ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK;
clock_system.sampling_frequency = 2250e6;
clock_system.low_jitter_mode_enabled = 1;
```

The code snippet below shows a use case where delay adjustment is enabled with a relative delay of 100 ps added to the external reference clock:

```
/* Use external reference clock with delay adjustment
   enabled and 100ps added delay. */
clock_system.reference_source = ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK;
clock_system.reference_frequency = 10e6;
clock_system.delay_adjustment_enabled = 1;
clock_system.delay_adjustment = 100e-12;
```

15.4.2 Input Routing

Changing the input routing parameters is disruptive because it resets the interface between ADC and FPGA, temporarily stopping the flow of data, and also affects several sections of the configuration phase `ADQParameters`, such as channel labels and front-end parameters.

The code snippet below shows an example where the input routing of API channel index 0 on an `ADQ32-1CH` digitizer is changed from the default input A to the alternate input B:

```
/* Allocate a variable to hold the clock system parameters. */
struct ADQInputRoutingParameters input_routing;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_INPUT_ROUTING,
                                      &input_routing);
if (result != sizeof(input_routing))
{
  /* Handle error */
}

/* Change the analog input connected to channel 0 from input A to input B. */
input_routing.channel[0].input = ADQ_ANALOG_INPUT_B;

/* Set up the clock system. */
result = ADQ_SetParameters(adq_cu, adq_num, &input_routing)
if (result != sizeof(input_routing))
{
  /* Handle error */
}
```

15.5 Configuration

In the configuration phase, the digitizer's parameters are given values that determine both the immediate behavior, but also the behavior during the acquisition phase (Section 15.6). An example of the former is the state of an output configured GPIO port when its value is changed. An example of the latter is the number of records to acquire for a target channel. The parameter values should be considered to be volatile information and will only persist as long as the digitizer is not reinitialized (Section 15.4). See Section 15.8 for details on interacting with the parameter space.

15.6 Acquisition

Acquiring data from the digitizer differs slightly depending on the configuration of the data transfer interface (Section 10). This section highlights the data readout interface, with records being transferred to the host computer's RAM. For other use cases, refer to the corresponding source code example. Assuming that the configuration step has been completed (Section 15.5) and the data transfer parameters have been configured according to Section 10.5. The data acquisition is started with

```
int result = ADQ_StartDataAcquisition(adq_cu, adq_num);
if (result != ADQ_EOK)
{
  /* Handle error */
}
```

If the call is successful, the data readout loop follows. The program waits for a record from any channel by using the constant `ADQ_ANY_CHANNEL` in the call to `WaitForRecordBuffer()`. Except for the value `ADQ_EAGAIN`, which indicates a timeout, negative values indicate errors where the data acquisition must be aborted.

```
bool done = false;
while (!done)
{
  struct ADQGen4Record *record;
  int channel = ADQ_ANY_CHANNEL;

  /* Wait for a record buffer with a 1000 ms timeout. */
  int64_t bytes_received = ADQ_WaitForRecordBuffer(
    adq_cu, adq_num, &channel, (void **)(&record), 1000, NULL
  );

  if (bytes_received == ADQ_EAGAIN)
  {
    /* Timeout */
    continue;
  }
  else if (bytes_received < 0)
  {
    /* An unexpected error, abort the acquisition. */
    break;
  }
```

At this point, the data processing step takes place. Refer to Section 10.5.3, step 5 for more information about important aspects to consider in this stage. Once the data has been processed, the API expects a call to `ReturnRecordBuffer()`, signaling that the underlying memory is once again available to place new data into. The data readout loop ends with evaluating the stop condition. This is highly application specific but common events include key presses or that a certain number of records have been acquired.

```
  result = ADQ_ReturnRecordBuffer(adq_cu, adq_num, channel, record);
  if (result != ADQ_EOK)
  {
    /* An unexpected error, abort the acquisition. */
    break;
  }

  /* Check for the stop condition. */
  done = ...
}
```

Finally, the phase ends with stopping the data acquisition process. This step should *always* be carried out regardless of if the program encountered an error or not.

```
result = ADQ_StopDataAcquisition(adq_cu, adq_num);
switch (result)
{
case ADQ_EOK:
case ADQ_EINTERRUPTED:
  /* Expected return value. */
  break;
default:
  /* Unexpected return value. */
  break;
}
```

15.7 Cleanup

In the cleanup phase the resources allocated by the user and the API should be deallocated. For the API, the cleanup is performed by calling

```
DeleteADQControlUnit(adq_cu);
```

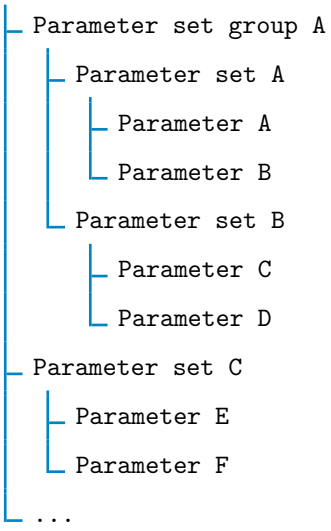
which will delete the control unit and all device objects. The memory allocated by the API, e.g. the record buffers, will also be freed. Any attempt to read this memory after the control unit has been deleted will cause access violations. To reconnect to the digitizer, the steps outlined in the identification phase (Section 15.3) must be executed.

15.8 Parameter Space

Both the initialization and configuration parts of the control flow use modifications to the digitizer's *parameter space* to configure and change the behavior of the digitizer.

The digitizer's parameter space is organized as a tree structure where the outermost nodes represent the individual parameters. These nodes are grouped together according to their function, forming a *parameter set*. Parameter sets with a common theme may also be grouped together, forming a *parameter set group*. The root node represents the digitizer itself, collecting all the parameter sets and parameter set groups to form a complete map of the digitizer's parameter space:

ADQ3 series digitizer



Interacting with the digitizer's parameters involves this tree structure and four functions:

- `InitializeParameters()`
- `GetParameters()`
- `ValidateParameters()`
- `SetParameters()`

From the perspective of the user application, these functions transport parameters in two directions:

- `InitializeParameters()` and `GetParameters()` are read operations, where parameter values flow from the API to the user application.
- `ValidateParameters()` and `SetParameters()` are write operations, where parameter values flow from the user application to the API.

The functions can operate on:

- the entire tree, effectively reading or writing the full parameter space with every operation; or
- parts of the tree, where only the targeted parameter set or parameter set group is affected.

Throughout the tree, there are specific nodes that hold information about the structure itself. These nodes signal that the tree may be broken off at this point and used together with the configuration functions, effectively configuring a subset of the parameter space. These specific nodes are the *parameter sets* and the *parameter set groups*.

The tree structure offers flexibility in choosing how to integrate the digitizer into the user application. In some cases, the most straight-forward way may be to operate on the full parameter set and keep the entire tree readily available. In other cases, the most reasonable approach may be to build up functionality around the various parameter sets, interacting with the digitizer on a more granular level.

15.8.1 In Practice

Most programming languages have a way of organizing data in a hierarchy. In the C programming language, *structures*, or *structs*, are used for this purpose. The header file contains the struct definitions that together build up the nodes of the parameter tree described in Section 15.8.

The result is a binary format that can be handled in the same way as any other similar structure: written to a file, read from a file, transmitted over a network, converted into a human readable format, parsed from a human readable format etc.

The code snippet below takes the approach of allocating a variable representing the full parameter space and calling `InitializeParameters()` to seed the parameters with their default values. The return value on success will be the size of the object while negative values indicate an error. This property is shared across all the configuration functions.

```
/* Allocate a variable to hold the digitizer parameters. */
struct ADQParameters adq;
int result = ADQ_InitializeParameters(adq_cu, adq_num, ADQ_PARAMETER_ID_TOP, &adq);
if (result != sizeof(adq))
{
  /* Handle error */
}
```

Following a successful initialization, it is safe to start modifying the parameter values. In the following code snippet, the data acquisition parameters for one of the channels are set up to acquire 10 records, each with 2048 samples every time the TRIG port detects a rising edge.

```
/* Activate one of the channels. */
adq.acquisition.channel[0].nof_records = 10;
adq.acquisition.channel[0].record_length = 2048;
adq.acquisition.channel[0].trigger_source = ADQ_EVENT_SOURCE_TRIG;
adq.acquisition.channel[0].trigger_edge = ADQ_EDGE_RISING;
```

It is important to note that technically, the digitizer state has *not* been modified at this point, only the values in the local parameter tree structure. Thus, since the application has not interacted with the hardware, there has been no need to handle errors as the assignments are processed.

The configuration is validated and written to the digitizer by calling `SetParameters()` with a reference to the updated parameter set. Since the full parameter set is passed to the function, the entire parameter space of the digitizer will be written, not just the acquisition parameters. The other parameters keep their default values assigned by the call to `InitializeParameters()`.

```
/* The other parameters get their default values from InitializeParameters(). */
result = ADQ_SetParameters(adq_cu, adq_num, &adq)
if (result != sizeof(adq))
{
  /* Handle error */
}
```

To only update the parameters of the data acquisition process, the tree can be split off at the `acquisition` node:

```
result = ADQ_SetParameters(adq_cu, adq_num, &adq.acquisition)
if (result != sizeof(adq.acquisition))
{
  /* Handle error */
}
```

15.8.2 JSON

To provide support for programming languages where the struct-based view of the parameter space is not easily represented in native data types—or just to generate a human-readable representation—an ASCII-based parameter API is also available. This API uses JSON-formatted text to represent the parameter space. Two sources/destinations for the JSON data are supported, each with their own set of API functions to interface with the digitizer:

- a zero terminated array of ASCII characters (a C-string):
  - `InitializeParametersString()`
  - `GetParametersString()`
  - `ValidateParametersString()`
  - `SetParametersString()`
- a file on the host computer:
  - `InitializeParametersFilename()`
  - `GetParametersFilename()`
  - `ValidateParametersFilename()`
  - `SetParametersFilename()`

Please refer to the corresponding entry in the API reference documentation (Appendix A) for more information.

Note

The JSON API is implemented as a translation layer between the text representation and the underlying binary format. All actions are executed using the binary format.

Important

Enumerations are represented as strings. The set of accepted values are the names defined by the corresponding enumeration.

Important

64-bit integers are represented as strings. Modifying these values using arithmetic operations requires conversion to/from a suitable native integer type in the employed programming language.

The code snippet below demonstrates how to read the current clock system parameters.

```
/* Allocate a character array large enough to hold the clock system parameter set. */
char clock_parameters[1024];
int result = ADQ_GetParametersString(
  adq_cu, adq_num, ADQ_PARAMETER_ID_CLOCK_SYSTEM, clock_parameters, 1024, 1
);

if (result < 0)
{
  /* Handle error */
}

/* Print the parameter set */
printf("%s", clock_parameters);
```

Which produces the output:

```
{
    "clock_generator": "ADQ_CLOCK_GENERATOR_INTERNAL_PLL",
    "reference_source": "ADQ_REFERENCE_CLOCK_SOURCE_INTERNAL",
    "sampling_frequency": 5000000000,
    "reference_frequency": 10000000,
    "delay_adjustment": 0,
    "low_jitter_mode_enabled": 1,
    "delay_adjustment_enabled": 0
}
```

16 Python API

A Python package named `pyadq` is provided for interfacing with the digitizer using Python. This package is a pure Python wrapper for the C API, and is the recommended way to access the digitizer in Python. To use the package, the following prerequisites have to be met:

- a system with Python 3.6 or later,
- the `numpy` and `ctypes` Python packages; and
- the ADQAPI.

The `pyadq` package contains two classes: `pyadq.ADQ` and `pyadq.ADQControlUnit` which wrap the ADQ object and the ADQ control unit object, respectively. Both of these classes have two types of member methods:

- Methods passed to directly to the ADQAPI. All of these methods are prefixed with either `ADQControlUnit_` or `ADQ_`, for example `pyadq.ADQ.ADQ_GetProductID`. These functions take the same arguments and return the same type as the ADQAPI call. All functions listed in this document are available under these prefixes.
- Methods implemented in Python which wrap one or multiple ADQAPI library calls, for example `pyadq.ADQ.SetParameters`. These functions have a more *pythonic* behavior. The complete list of these functions can be found in the package files `ADQ.py` and `ADQControlUnit.py` files.

All of the C structs listed in Section A.3 have a corresponding Python implementation with native Python types. The configuration functions:

- `pyadq.ADQ.SetParameters`,
- `pyadq.ADQ.GetParameters`,
- `pyadq.ADQ.InitializeParameters`; and
- `pyadq.ADQ.ValidateParameters`

all use the native Python implementations of the C structs.

The configuration functions for the JSON API described in Section 15.8.2 also have Python wrappers to help with the string handling:

- `pyadq.ADQ.SetParametersString`
- `pyadq.ADQ.GetParametersString`
- `pyadq.ADQ.InitializeParametersString`
- `pyadq.ADQ.ValidateParametersString`
- `pyadq.ADQ.SetParametersFilename`
- `pyadq.ADQ.GetParametersFilename`
- `pyadq.ADQ.InitializeParametersFilename`
- `pyadq.ADQ.ValidateParametersFilename`

16.1 Installation

The `pyadq` package can be installed to the directory for user-installed packages by executing

```
$ pip install --user <path to pyadq>
```

References

[1] Teledyne Signal Processing Devices Sweden AB, *22-2869 ADQ30 datasheet*. Technical Specification.

[2] Teledyne Signal Processing Devices Sweden AB, *20-2378 ADQ32 datasheet*. Technical Specification.

[3] Teledyne Signal Processing Devices Sweden AB, *20-2451 ADQ33 datasheet*. Technical Specification.

[4] Teledyne Signal Processing Devices Sweden AB, *20-2540 ADQ36 datasheet*. Technical Specification.

[5] Teledyne Signal Processing Devices Sweden AB, *18-2059 ADQUpdater User Guide*. Technical Manual.

[6] Teledyne Signal Processing Devices Sweden AB, *15-1494 Digitizer OS Support*. Technical Specification.

[7] Teledyne Signal Processing Devices Sweden AB, *14-1351 ADQAPI Reference Guide*. Technical Manual.

[8] Teledyne Signal Processing Devices Sweden AB, *20-2507 ADQ3 Series FWDAQ Development Kit User Guide*. Technical Manual.

A API Reference

This section contains detailed descriptions of the defines, enumerations, structures and functions that together make up the API for ADQ3 series digitizers.

Important

All objects described in the following sections are defined in the `ADQAPI.h` header file. Please refrain from redefining constants and structures.

A.1 Defines

This section lists the constants that are not tied to a particular type. Many of the constants are on the form of `ADQ_MAX_NOF...` and define absolute limits of what the API supports with respect to various objects. These limits are set so that they are not a problem in practice, e.g. no digitizer will feature more channels than the value of `ADQ_MAX_NOF_CHANNELS`.

The primary use of the constants are as an aid in writing robust user applications that automatically adapt to whichever API version it is compiled against. For example, the structures defined in Section A.3 use these constants to define upper bounds for their static arrays.

```
#define ADQAPI_VERSION_MAJOR 9
```

Description

The major version number of the API. This constant is used with the function ADQAPI_ValidateVersion() to protect against dynamically linking against an incompatible API. It is strongly recommended to implement this safe-guard in the user application.

```
#define ADQAPI_VERSION_MINOR 0
```

Description

The minor version number of the API. The documentation for ADQAPI_VERSION_MAJOR applies for this constant as well.

```
#define ADQ_MAX_NOF_CHANNELS 8
```

Description

The maximum number of channels supported by the API.

```
#define ADQ_MAX_NOF_BUFFERS 16
```

Description

The maximum number of buffers supported by the API.

```
#define ADQ_MAX_NOF_PORTS 8
```

Description

The maximum number of ports supported by the API.

```
#define ADQ_MAX_NOF_PINS 16
```

Description

The maximum number of pins supported by a port.

```
#define ADQ_MAX_NOF_ADC_CORES 4
```

Description

The maximum number of ADC cores supported by a channel.

```
#define ADQ_MAX_NOF_INPUT_RANGES 8
```

Description

The maximum number of input range configurations supported by a channel.

```
#define ADQ_MAX_NOF_PATTERN_GENERATORS 2
```

Description

The maximum number of pattern generators.

```
#define ADQ_MAX_NOF_PULSE_GENERATORS 4
```

Description

The maximum number of pulse generators.

```
#define ADQ_MAX_NOF_PATTERN_INSTRUCTIONS 16
```

Description

The maximum number of pattern generator instructions.

```
#define ADQ_MAX_NOF_TEMPERATURE_SENSORS 16
```

Description

The maximum number of temperature sensors.

```
#define ADQ_MAX_NOF_FILTER_COEFFICIENTS 10
```

Description

The maximum number of filter coefficients.

```
#define ADQ_MAX_NOF_MATRIX_INPUTS 8
```

Description

The maximum number of inputs to the matrix event source.

```
#define ADQ_MAX_NOF_PDRX_EQUALIZER_COEFFICIENTS 49
```

Description

The maximum number of coefficients supported by the PDRX equalizer.

```
#define ADQ_MAX_NOF_PDRX_REFLECTION_FILTER_COEFFICIENTS 9
```

Description

The maximum number of coefficients supported by the PDRX reflection filter.

```
#define ADQ_MAX_NOF_ATD_THRESHOLD_FILTER_COEFFICIENTS 9
```

Description

The maximum number of coefficients supported by the ATD signal processing module threshold filter.

```
#define ADQ_MAX_NOF_PLLS 8
```

Description

The maximum number of PLLs in the digitizer's clock system.

```
#define ADQ_ANY_CHANNEL (-1)
```

Description

An alias for the value −1 which causes `WaitForRecordBuffer()` to return the first available data from *any channel* when used by its `channel` parameter.

```
#define ADQ_INFINITE_RECORD_LENGTH (-1)
```

Description

An alias for the value −1 which causes the data acquisition process (Section 9) to become unbounded in terms of the length of the acquired record when passed to the parameter `record_length`.

```
#define ADQ_INFINITE_NOF_RECORDS (-1)
```

Description

An alias for the value −1 which causes the data acquisition process (Section 9) to become unbounded in terms of the number of records to acquire when passed to the parameter `nof_records`.

```
#define ADQ_UNITY_GAIN (1024)
```

Description

An alias for the value 1024 which signifies *unity gain* in the context of the digital gain and offset module (Section 5.1).

```
#define ADQ_FOLLOW_RECORD_TRANSFER_BUFFER (-1)
```

Description

An alias for the value −1 which is intended to be passed to the data readout parameter `nof_record_buffers_in_array` to specify that the array should follow the capacity of the underlying record transfer buffer w.r.t. the number of array elements.

```
#define ADQ_PARAMETERS_MAGIC (0xAA559977AA559977ull)
```

Description

A magic number to indicate the end of a parameter struct. This constant should never appear in the user application if the recommended configuration flow is followed. It is managed by the two configuration functions `InitializeParameters()` and `GetParameters()`.

```
#define ADQ_DATA_READOUT_STATUS_FLAGS_OK (0)
```

Description

An alias for the value 0 which is used to indicate that the data readout status member `flags` reports no errors.

```
#define ADQ_DATA_READOUT_STATUS_FLAGS_STARVING (1u << 0)
```

Description

An alias for the value `1u << 0` which is used to indicate that the data readout status member `flags` reports that interface is starving for record buffers.

```
#define ADQ_DATA_READOUT_STATUS_FLAGS_INCOMPLETE (1u << 1)
```

Description

An alias for the value `1u << 1` which may be used to mask the data readout status member `flags`. If the result is nonzero, the record buffer emitted by `WaitForRecordBuffer()` contains incomplete record data. See Section 10.5.7 for details.

```
#define ADQ_DATA_READOUT_STATUS_FLAGS_DISCARDED (1u << 2)
```

Description

An alias for the value `1u << 2` which may be used to mask the data readout status member `flags`. If the result is nonzero, a record was discarded due to incomplete data. See Section 10.5.6 for additional information.

```
#define LOG_LEVEL_ERROR 0
```

Description

An alias for the value 0 which enables *error* logging when passed to `ADQControlUnit_EnableErrorTrace()`.

```
#define LOG_LEVEL_WARN 1
```

Description

An alias for the value 1 which enables *error and warning* logging when passed to `ADQControlUnit_EnableErrorTrace()`.

```
#define LOG_LEVEL_INFO 2
```

Description

An alias for the value 2 which enables *error, warning and info* logging when passed to `ADQControlUnit_EnableErrorTrace()`.

```
#define ADQ_DATA_FORMAT_INT16 0
```

Description

An alias for the value 0 which is used by the record header field `data_format` to indicate that the record contains 16-bit, 2's complement data.

```
#define ADQ_DATA_FORMAT_INT32 1
```

Description

An alias for the value 1 which is used by the record header field `data_format` to indicate that the record contains 32-bit, 2's complement data.

```
#define ADQ_DATA_FORMAT_PULSE_ATTRIBUTES 3
```

Description

An alias for the value 3 which is used by the record header field `data_format` to indicate that the record contains pulse attribute data. This value is only possible when the digitizer is running the FWPD firmware (Section 5.7).

```
#define ADQ_RECORD_STATUS_OVERFLOW (1u << 0)
```

Description

An alias for the value `1 << 0` which may be used to mask the header field `record_status`. If the result is nonzero, the record is incomplete and has lost data at the end in an unrecoverable way due to an overflow. See Section 10.6 for more information.

```
#define ADQ_RECORD_STATUS_OVERRANGE (1u << 2)
```

Description

An alias for the value `1 << 2` which may be used to mask the header field `record_status`. If the result is nonzero, one or several samples in the record have saturated at the maximum or minimum value when the actual value cannot be represented in the available range.

```
#define ADQ_RECORD_STATUS_RISING_EDGE (1u << 3)
```

Description

An alias for the value `1 << 3` which may be used to mask the header field `record_status`. If the result is nonzero, the record was triggered by a rising edge event from the target `trigger_source`. This is useful to differentiate between records when the `trigger_edge` is set to `ADQ_EDGE_BOTH`.

```
#define ADQ_PULSE_ATTRIBUTES_STATUS_VALID (1u << 0)
```

Description

An alias for the value `1 << 0` which may be used to mask the pulse attributes `status` field. If the result is nonzero, all attributes are valid. If the result is zero, one or several of the attributes are invalid. See Section 5.7.4 for more information.

A.2 Enumerations

This section lists the constants that are tied to a particular type, i.e. enumerations. To improve readability and to make the source code easier to maintain, it is strongly recommended to use these named constants and not the corresponding integer value.

Note

The enumerations types defined in this section are made as 32-bit types.

```
enum ADQParameterId {
    ADQ_PARAMETER_ID_RESERVED = 0,
    ADQ_PARAMETER_ID_DATA_ACQUISITION = 1,
    ADQ_PARAMETER_ID_DATA_TRANSFER = 2,
    ADQ_PARAMETER_ID_DATA_READOUT = 3,
    ADQ_PARAMETER_ID_CONSTANT = 4,
    ADQ_PARAMETER_ID_DIGITAL_GAINANDOFFSET = 5,
    ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL = 6,
    ADQ_PARAMETER_ID_DBS = 7,
    ADQ_PARAMETER_ID_SAMPLE_SKIP = 8,
    ADQ_PARAMETER_ID_TEST_PATTERN = 9,
    ADQ_PARAMETER_ID_EVENT_SOURCE_PERIODIC = 10,
    ADQ_PARAMETER_ID_EVENT_SOURCE_TRIG = 11,
    ADQ_PARAMETER_ID_EVENT_SOURCE_SYNC = 12,
    ADQ_PARAMETER_ID_ANALOG_FRONTEND = 13,
    ADQ_PARAMETER_ID_PATTERN_GENERATOR0 = 14,
    ADQ_PARAMETER_ID_PATTERN_GENERATOR1 = 15,
    ADQ_PARAMETER_ID_EVENT_SOURCE = 16,
    ADQ_PARAMETER_ID_SIGNAL_PROCESSING = 17,
    ADQ_PARAMETER_ID_FUNCTION = 18,
    ADQ_PARAMETER_ID_TOP = 19,
    ADQ_PARAMETER_ID_PORT_TRIG = 20,
    ADQ_PARAMETER_ID_PORT_SYNC = 21,
    ADQ_PARAMETER_ID_PORT_SYNC0 = 22,
    ADQ_PARAMETER_ID_PORT_SYNCI = 23,
    ADQ_PARAMETER_ID_PORT_CLK = 24,
    ADQ_PARAMETER_ID_PORT_CLKI = 25,
    ADQ_PARAMETER_ID_PORT_CLKO = 26,
    ADQ_PARAMETER_ID_PORT_GPIOA = 27,
    ADQ_PARAMETER_ID_PORT_GPIOB = 28,
    ADQ_PARAMETER_ID_PORT_PXIE = 29,
    ADQ_PARAMETER_ID_PORT_MTCA = 30,
    ADQ_PARAMETER_ID_PULSE_GENERATOR0 = 31,
    ADQ_PARAMETER_ID_PULSE_GENERATOR1 = 32,
    ADQ_PARAMETER_ID_PULSE_GENERATOR2 = 33,
    ADQ_PARAMETER_ID_PULSE_GENERATOR3 = 34,
    ADQ_PARAMETER_ID_TIMESTAMP_SYNCHRONIZATION = 35,
    ADQ_PARAMETER_ID_FIR_FILTER = 36,
    ADQ_PARAMETER_ID_PORT_GPIOC = 37,
    ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOA = 38,
    ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOB = 39,
    ADQ_PARAMETER_ID_CLOCK_SYSTEM = 40,
    ADQ_PARAMETER_ID_EVENT_SOURCE_PXIE = 41,
```

```
    ADQ_PARAMETER_ID_EVENT_SOURCE_SOFTWARE = 42,
    ADQ_PARAMETER_ID_EVENT_SOURCE_MATRIX = 43,
    ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL_MATRIX = 44,
    ADQ_PARAMETER_ID_PDRX = 45,
    ADQ_PARAMETER_ID_ATD = 46,
    ADQ_PARAMETER_ID_DAISY_CHAIN = 47,
    ADQ_PARAMETER_ID_INPUT_ROUTING = 48,
    ADQ_PARAMETER_ID_PD = 49
}
```

Description

An enumeration of the parameter set identification numbers used by the configuration functions `InitializeParameters()`, `GetParameters()`, `SetParameters()` and `ValidateParameters()`.

Values

`ADQ_PARAMETER_ID_RESERVED (0)`
: Reserved

`ADQ_PARAMETER_ID_DATA_ACQUISITION (1)`
: The identification number for the data acquisition parameters, defined by `ADQDataAcquisitionParameters`.

`ADQ_PARAMETER_ID_DATA_TRANSFER (2)`
: The identification number for the data transfer parameters, defined by `ADQDataTransferParameters`.

`ADQ_PARAMETER_ID_DATA_READOUT (3)`
: The identification number for the data readout parameters, defined by `ADQDataReadoutParameters`.

`ADQ_PARAMETER_ID_CONSTANT (4)`
: The identification number for the constant parameters, defined by `ADQConstantParameters`.

`ADQ_PARAMETER_ID_DIGITAL_GAINANDOFFSET (5)`
: The identification number for the digitial gain and offset parameters, defined by `ADQDigitalGainAndOffsetParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL (6)`
: The identification number for the level event source parameters, defined by `ADQEventSourceLevelParameters`.

`ADQ_PARAMETER_ID_DBS (7)`
: The identification number for the digital baseline stabilization parameters, defined by `ADQDbsParameters`.

`ADQ_PARAMETER_ID_SAMPLE_SKIP (8)`

The identification number for the sample skip parameters, defined by `ADQSampleSkipParameters`.

`ADQ_PARAMETER_ID_TEST_PATTERN (9)`

The identification number for the constant parameters, defined by `ADQConstantParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE_PERIODIC (10)`

The identification number for the test pattern parameters, defined by `ADQTestPatternParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE_TRIG (11)`

The identification number for the parameters of the event source associated with the TRIG port, defined by `ADQEventSourcePortParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE_SYNC (12)`

The identification number for the parameters of the event source associated with the SYNC port, defined by `ADQEventSourcePortParameters`.

`ADQ_PARAMETER_ID_ANALOG_FRONTEND (13)`

The identification number for the analog front-end parameters, defined by `ADQAnalogFrontend-Parameters`.

`ADQ_PARAMETER_ID_PATTERN_GENERATOR0 (14)`

The identification number for the parameters of the first pattern generator instance, defined by `ADQPatternGeneratorParameters`.

`ADQ_PARAMETER_ID_PATTERN_GENERATOR1 (15)`

The identification number for the parameters of the second pattern generator instance, defined by `ADQPatternGeneratorParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE (16)`

The identification number for the parameters of *all* the event sources, defined by `ADQEventSource-Parameters`.

`ADQ_PARAMETER_ID_SIGNAL_PROCESSING (17)`

The identification number for the parameters of *all* the signal processing modules, defined by `ADQSignalProcessingParameters`.

`ADQ_PARAMETER_ID_FUNCTION (18)`

The identification number for the parameters of *all* the function modules, defined by `ADQFunction-Parameters`.

`ADQ_PARAMETER_ID_TOP (19)`

The identification number for the structure representing the entire parameter space of the digitizer, defined by `ADQParameters`.

`ADQ_PARAMETER_ID_PORT_TRIG (20)`

The identification number for the parameters associated with the TRIG port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_SYNC (21)`

The identification number for the parameters associated with the SYNC port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_SYNCO (22)`

The identification number for the parameters associated with the SYNCO port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_SYNCI (23)`

The identification number for the parameters associated with the SYNCI port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_CLK (24)`

The identification number for the parameters associated with the CLK port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_CLKI (25)`

The identification number for the parameters associated with the CLKI port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_CLKO (26)`

The identification number for the parameters associated with the CLKO port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_GPIOA (27)`

The identification number for the parameters associated with the GPIOA port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_GPIOB (28)`

The identification number for the parameters associated with the GPIOB port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_PXIE (29)`

The identification number for the parameters associated with the PXIE port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PORT_MTCA (30)`

The identification number for the parameters associated with the MTCA port, defined by `ADQPortParameters`.

`ADQ_PARAMETER_ID_PULSE_GENERATOR0 (31)`

The identification number for the parameters of the first pulse generator, defined by `ADQPulse-`

GeneratorParameters.

`ADQ_PARAMETER_ID_PULSE_GENERATOR1 (32)`

The identification number for the parameters of the second pulse generator, defined by ADQPulseGeneratorParameters.

`ADQ_PARAMETER_ID_PULSE_GENERATOR2 (33)`

The identification number for the parameters of the third pulse generator, defined by ADQPulseGeneratorParameters.

`ADQ_PARAMETER_ID_PULSE_GENERATOR3 (34)`

The identification number for the parameters of the fourth pulse generator, defined by ADQPulseGeneratorParameters.

`ADQ_PARAMETER_ID_TIMESTAMP_SYNCHRONIZATION (35)`

The identification number for the parameters of the timestamp synchronization defined by ADQTimestampSynchronizationParameters.

`ADQ_PARAMETER_ID_FIR_FILTER (36)`

The identification number for the parameters of the FIR filter defined by ADQFirFilterParameters.

`ADQ_PARAMETER_ID_PORT_GPIOC (37)`

The identification number for the parameters associated with the GPIOC port, defined by ADQPortParameters.

`ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOA (38)`

The identification number for the parameters of the event source associated with the GPIOA port, defined by ADQEventSourcePortParameters.

`ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOB (39)`

The identification number for the parameters of the event source associated with the GPIOB port, defined by ADQEventSourcePortParameters.

`ADQ_PARAMETER_ID_CLOCK_SYSTEM (40)`

The identification number for the parameters of the clock system defined by ADQClockSystemParameters.

`ADQ_PARAMETER_ID_EVENT_SOURCE_PXIE (41)`

The identification number for the parameters of the event source associated with the PXIE port, defined by ADQEventSourcePortParameters.

`ADQ_PARAMETER_ID_EVENT_SOURCE_SOFTWARE (42)`

The identification number for the parameters of the software controlled event source, defined by ADQEventSourceSoftwareParameters.

`ADQ_PARAMETER_ID_EVENT_SOURCE_MATRIX (43)`

The identification number for the parameters of the matrix event source, defined by `ADQEventSourceMatrixParameters`.

`ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL_MATRIX (44)`

The identification number for the parameters of the signal level event source matrix, defined by `ADQEventSourceLevelMatrixParameters`.

`ADQ_PARAMETER_ID_PDRX (45)`

The identification number for the parameters of the pulse detection range extension module, defined by `ADQPdrxParameters`.

`ADQ_PARAMETER_ID_ATD (46)`

The identification number for the parameters of the ATD signal processing module, defined by `ADQAtdParameters`.

`ADQ_PARAMETER_ID_DAISY_CHAIN (47)`

The identification number for the parameters of the daisy chain function, defined by `ADQDaisyChainParameters`.

`ADQ_PARAMETER_ID_INPUT_ROUTING (48)`

The identification number for the parameters that determine routing of analog inputs to channels in the digitizer, defined by `ADQInputRoutingParameters`.

`ADQ_PARAMETER_ID_PD (49)`

The identification number for the parameters of the PD signal processing module, defined by `ADQPdParameters`.

```
enum ADQStatusId {
    ADQ_STATUS_ID_RESERVED                  = 0,
    ADQ_STATUS_ID_OVERFLOW                  = 1,
    ADQ_STATUS_ID_DRAM                      = 2,
    ADQ_STATUS_ID_ACQUISITION               = 3,
    ADQ_STATUS_ID_TEMPERATURE               = 4,
    ADQ_STATUS_ID_CLOCK_SYSTEM              = 5,
    ADQ_STATUS_ID_TIMESTAMP_SYNCHRONIZATION = 6,
    ADQ_STATUS_ID_DAISY_CHAIN               = 7,
    ADQ_STATUS_ID_LICENSE                   = 8
}
```

Description

An enumeration of the identification numbers used by the status query function `GetStatus()`.

Values

`ADQ_STATUS_ID_RESERVED (0)`

Reserved

`ADQ_STATUS_ID_OVERFLOW (1)`

The identification number for the overflow status, defined by `ADQOverflowStatus`.

`ADQ_STATUS_ID_DRAM (2)`

The identification number for the DRAM status, defined by `ADQDramStatus`.

`ADQ_STATUS_ID_ACQUISITION (3)`

The identification number for the data acquisition status, defined by `ADQAcquisitionStatus`.

`ADQ_STATUS_ID_TEMPERATURE (4)`

The identification number for the status of the temperature sensors, defined by `ADQTemperature-Status`.

`ADQ_STATUS_ID_CLOCK_SYSTEM (5)`

The identification number for the status of the clock system, defined by `ADQClockSystemStatus`.

`ADQ_STATUS_ID_TIMESTAMP_SYNCHRONIZATION (6)`

The identification number for the status of the timestamp synchronization, defined by `ADQTimestampSynchronizationStatus`.

`ADQ_STATUS_ID_DAISY_CHAIN (7)`

The identification number for the status of the daisy chain, defined by `ADQDaisyChainStatus`.

`ADQ_STATUS_ID_LICENSE (8)`

The identification number for the status of the digitizer's licenses, defined by `ADQLicenseStatus`.

```
enum ADQEventSource {
    ADQ_EVENT_SOURCE_INVALID                = 0,
    ADQ_EVENT_SOURCE_SOFTWARE               = 1,
    ADQ_EVENT_SOURCE_TRIG                   = 2,
    ADQ_EVENT_SOURCE_LEVEL                  = 3,
    ADQ_EVENT_SOURCE_PERIODIC               = 4,
    ADQ_EVENT_SOURCE_PXIE_STARB             = 6,
    ADQ_EVENT_SOURCE_TRIG2                  = 7,
    ADQ_EVENT_SOURCE_TRIG3                  = 8,
    ADQ_EVENT_SOURCE_SYNC                   = 9,
    ADQ_EVENT_SOURCE_MTCA_MLVDS             = 10,
    ADQ_EVENT_SOURCE_TRIG_GATED_SYNC        = 11,
    ADQ_EVENT_SOURCE_TRIG_CLKREF_SYNC       = 12,
    ADQ_EVENT_SOURCE_MTCA_MLVDS_CLKREF_SYNC = 13,
    ADQ_EVENT_SOURCE_PXI_TRIG               = 14,
    ADQ_EVENT_SOURCE_PXIE_STARB_CLKREF_SYNC = 16,
    ADQ_EVENT_SOURCE_SYNC_CLKREF_SYNC       = 19,
    ADQ_EVENT_SOURCE_DAISY_CHAIN            = 23,
    ADQ_EVENT_SOURCE_SOFTWARE_CLKREF_SYNC   = 24,
    ADQ_EVENT_SOURCE_GPIOA0                 = 25,
    ADQ_EVENT_SOURCE_GPIOA1                 = 26,
    ADQ_EVENT_SOURCE_GPIOB0                 = 60,
    ADQ_EVENT_SOURCE_PXIE_TRIG0             = 90,
    ADQ_EVENT_SOURCE_PXIE_TRIG1             = 91,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL0         = 100,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL1         = 101,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL2         = 102,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL3         = 103,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL4         = 104,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL5         = 105,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL6         = 106,
    ADQ_EVENT_SOURCE_LEVEL_CHANNEL7         = 107,
    ADQ_EVENT_SOURCE_REFERENCE_CLOCK        = 120,
    ADQ_EVENT_SOURCE_MATRIX                 = 121,
    ADQ_EVENT_SOURCE_LEVEL_MATRIX           = 122
}
```

Description

An enumeration of the event sources which can be utilized by various functions of the digitizer, e.g. as a source for trigger events in the data acquisition process (see Section 9). Not all digitizer models support all the event sources. Refer to the ADQAPI reference guide [7] for more information.

Values

`ADQ_EVENT_SOURCE_INVALID (0)`

The invalid event source. This constant is commonly used to signal the absence of an event source, often implying that the function is disabled.

`ADQ_EVENT_SOURCE_SOFTWARE (1)`

The software-controlled event source. A software event is issued by calling the function `SWTrig()`. It is *not* possible to guarantee or control the timing of the issued software event.

`ADQ_EVENT_SOURCE_TRIG (2)`

The event source associated with the TRIG port. The parameters for this event source is defined by `ADQEventSourcePortParameters` with the struct identifier `ADQ_PARAMETER_ID_EVENT_SOURCE_TRIG`.

`ADQ_EVENT_SOURCE_LEVEL (3)`

This constant specifies the signal level event source (Section 6.4) analyzing data from the channel itself and is *only* applicable in contexts where its use is not ambiguous. For example, the channel-specific data acquisition parameter `trigger_source` may be set to this value, in which case these two lines:

```
channel[0].trigger_source = ADQ_EVENT_SOURCE_LEVEL;
channel[1].trigger_source = ADQ_EVENT_SOURCE_LEVEL;
```

are equivalent to:

```
channel[0].trigger_source = ADQ_EVENT_SOURCE_LEVEL_CHANNEL0;
channel[1].trigger_source = ADQ_EVENT_SOURCE_LEVEL_CHANNEL1;
```

The parameters of these event sources are defined by `ADQEventSourceLevelParameters`.

`ADQ_EVENT_SOURCE_PERIODIC (4)`

The periodic event source, see Section 6.3 for more details. The parameters are defined by `ADQEventSourcePeriodicParameters`.

`ADQ_EVENT_SOURCE_PXIE_STARB (6)`

The event source associated with the STARB signal in the PXIe port. Only digitizers with a PXIe interface supports this event source.

`ADQ_EVENT_SOURCE_TRIG2 (7)`

Reserved

`ADQ_EVENT_SOURCE_TRIG3 (8)`

Reserved

`ADQ_EVENT_SOURCE_SYNC (9)`

The event source associated with the SYNC port. The parameters for this event source is defined by `ADQEventSourcePortParameters` with the struct identifier `ADQ_PARAMETER_ID_EVENT_SOURCE_`

SYNC.

ADQ_EVENT_SOURCE_MTCA_MLVDS (10)
: Reserved

ADQ_EVENT_SOURCE_TRIG_GATED_SYNC (11)
: Reserved

ADQ_EVENT_SOURCE_TRIG_CLKREF_SYNC (12)
: Reserved

ADQ_EVENT_SOURCE_MTCA_MLVDS_CLKREF_SYNC (13)
: Reserved

ADQ_EVENT_SOURCE_PXI_TRIG (14)
: Reserved

ADQ_EVENT_SOURCE_PXIE_STARB_CLKREF_SYNC (16)
: Reserved

ADQ_EVENT_SOURCE_SYNC_CLKREF_SYNC (19)
: Reserved

ADQ_EVENT_SOURCE_DAISY_CHAIN (23)
: Reserved

ADQ_EVENT_SOURCE_SOFTWARE_CLKREF_SYNC (24)
: Reserved

ADQ_EVENT_SOURCE_GPIOA0 (25)
: The event source associated with pin 0 of the GPIOA port. This port may be labelled differently on the digitizer's front panel. This event source is not associated with any parameters. It senses a digital signal with the event threshold set to halfway between the logic low and logic high levels.

ADQ_EVENT_SOURCE_GPIOA1 (26)
: Reserved

ADQ_EVENT_SOURCE_GPIOB0 (60)
: The event source associated with pin 0 of the GPIOB port. This port may be labelled differently on the digitizer's front panel. This event source is not associated with any parameters. It senses a digital signal with the event threshold set to halfway between the logic low and logic high levels.

ADQ_EVENT_SOURCE_PXIE_TRIG0 (90)
: The event source associated with the TRIG0 signal in the PXIe port. Only digitizers with a PXIe interface supports this event source.

`ADQ_EVENT_SOURCE_PXIE_TRIG1 (91)`

The event source associated with the TRIG1 signal in the PXIe port. Only digitizers with a PXIe interface supports this event source.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL0 (100)`

The signal level event source analyzing data from channel 0. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL1 (101)`

The signal level event source analyzing data from channel 1. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL2 (102)`

The signal level event source analyzing data from channel 2. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL3 (103)`

The signal level event source analyzing data from channel 3. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL4 (104)`

The signal level event source analyzing data from channel 4. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL5 (105)`

The signal level event source analyzing data from channel 5. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL6 (106)`

The signal level event source analyzing data from channel 6. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_LEVEL_CHANNEL7 (107)`

The signal level event source analyzing data from channel 7. See Section 6.4 for more information.

`ADQ_EVENT_SOURCE_REFERENCE_CLOCK (120)`

The event source associated with the reference clock. Only the rising edge, i.e. `ADQ_EDGE_RISING`, is supported by this event source. See Section 6.11 for more information.

`ADQ_EVENT_SOURCE_MATRIX (121)`

The event source used for triggering on a combination of the other event sources. Each source can be individually configured to trigger on rising, falling or both edges, given that the underlying event source supports those edges. See Section 6.10 for more information.

`ADQ_EVENT_SOURCE_LEVEL_MATRIX (122)`

The dedicated event source used to combine signal level events into a new event stream. For example, the event source can be configured to output the rising edge events from channel A together with both edge events from channel B. See Section 6.5 for more information.

```
enum ADQTestPatternSource {
    ADQ_TEST_PATTERN_SOURCE_DISABLE                     = 0,
    ADQ_TEST_PATTERN_SOURCE_COUNT_UP                    = 1,
    ADQ_TEST_PATTERN_SOURCE_COUNT_DOWN                  = 2,
    ADQ_TEST_PATTERN_SOURCE_TRIANGLE                    = 3,
    ADQ_TEST_PATTERN_SOURCE_PULSE                       = 4,
    ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_WIDTH            = 5,
    ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_AMPLITUDE        = 6,
    ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_WIDTH_AMPLITUDE  = 7,
    ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE                 = 8,
    ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_WIDTH      = 9,
    ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_AMPLITUDE  = 10,
    ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_WIDTH_AMPLITUDE = 11
}
```

Description

An enumeration of the test pattern sources which can be used by the test pattern module (Section 11) to replace the ADC data for a target channel.

Values

`ADQ_TEST_PATTERN_SOURCE_DISABLE (0)`

The test pattern generator is disabled.

`ADQ_TEST_PATTERN_SOURCE_COUNT_UP (1)`

A counter test pattern source with an upwards direction. The ADC data is replaced with a positive sawtooth wave, wrapping to the largest negative value on overflow.

`ADQ_TEST_PATTERN_SOURCE_COUNT_DOWN (2)`

A counter test pattern source with a downwards direction. The ADC data is replaced with a negative sawtooth wave, wrapping to the largest positive value on underflow.

`ADQ_TEST_PATTERN_SOURCE_TRIANGLE (3)`

A counter test pattern source where the direction is alternating, inverting at the extreme values in the vertical range. In other words: the ADC data is replaced with a triangle wave.

`ADQ_TEST_PATTERN_SOURCE_PULSE (4)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_WIDTH (5)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_AMPLITUDE (6)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_PRBS_WIDTH_AMPLITUDE (7)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE (8)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_WIDTH (9)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_AMPLITUDE (10)`

Reserved

`ADQ_TEST_PATTERN_SOURCE_PULSE_NOISE_PRBS_WIDTH_AMPLITUDE (11)`

Reserved

```
enum ADQPort {
    ADQ_PORT_TRIG  = 0,
    ADQ_PORT_SYNC  = 1,
    ADQ_PORT_SYNCO = 2,
    ADQ_PORT_SYNCI = 3,
    ADQ_PORT_CLK   = 4,
    ADQ_PORT_CLKI  = 5,
    ADQ_PORT_CLKO  = 6,
    ADQ_PORT_GPIOA = 7,
    ADQ_PORT_GPIOB = 8,
    ADQ_PORT_PXIE  = 9,
    ADQ_PORT_MTCA  = 10,
    ADQ_PORT_GPIOC = 11
}
```

Description

An enumeration of the digitizer's ports (Section 8). Not all digitizer models feature every port in the list. This enumeration is also intended to make the indexing of the `port` array more readable. For example, it is recommended to write

```
struct ADQParameters adq;
adq.port[ADQ_PORT_SYNC].pin[0].direction = ADQ_DIRECTION_OUT;
adq.port[ADQ_PORT_SYNC].pin[0].function = ADQ_FUNCTION_GPIO;
adq.port[ADQ_PORT_SYNC].pin[0].invert_output = 0;
adq.port[ADQ_PORT_SYNC].pin[0].value = 1;
```

rather than

```
struct ADQParameters adq;
adq.port[1].pin[0].direction = ADQ_DIRECTION_OUT;
adq.port[1].pin[0].function = ADQ_FUNCTION_GPIO;
adq.port[1].pin[0].invert_output = 0;
adq.port[1].pin[0].value = 1;
```

Values

`ADQ_PORT_TRIG (0)`
: A constant specifying the TRIG port.

`ADQ_PORT_SYNC (1)`
: A constant specifying the SYNC port.

`ADQ_PORT_SYNCO (2)`
: A constant specifying the SYNCO port. Unused on ADQ3 series digitizers.

`ADQ_PORT_SYNCI (3)`
: A constant specifying the SYNCI port. Unused on ADQ3 series digitizers.

`ADQ_PORT_CLK (4)`
: A constant specifying the CLK port.

`ADQ_PORT_CLKI (5)`
: A constant specifying the CLKI port. Unused on ADQ3 series digitizers.

`ADQ_PORT_CLKO (6)`
: A constant specifying the CLKO port. Unused on ADQ3 series digitizers.

`ADQ_PORT_GPIOA (7)`
: The GPIOA port. On ADQ32 and ADQ33, this port is labeled “GPIO” on the front panel.

`ADQ_PORT_GPIOB (8)`
: The GPIOB port. Refer to the product datasheet for pin mapping.

`ADQ_PORT_PXIE (9)`
: A constant specifying the PXIe port. The pins in this port are named, see `ADQPinPxie` for additional details.

`ADQ_PORT_MTCA (10)`
: A constant specifying the MTCA port. Unused on ADQ3 series digitizers.

`ADQ_PORT_GPIOC (11)`
: The GPIOC port. Refer to the product datasheet for pin mapping.

```
enum ADQPinPxie {
    ADQ_PIN_PXIE_TRIG0 = 0,
    ADQ_PIN_PXIE_TRIG1 = 1,
    ADQ_PIN_PXIE_STARA = 2,
    ADQ_PIN_PXIE_STARB = 3,
    ADQ_PIN_PXIE_STARC = 4
}
```

Description

An enumeration of the pins in the PXIe port. Only digitizers with the PXIe interface have this port. This enumeration is intended to make the indexing of the `pin` array more readable. For example, it is recommended to write

```
struct ADQParameters adq;
adq.port[ADQ_PORT_PXIE].pin[ADQ_PIN_PXIE_STARC].direction = ADQ_DIRECTION_OUT;
adq.port[ADQ_PORT_PXIE].pin[ADQ_PIN_PXIE_STARC].function = ADQ_FUNCTION_GPIO;
adq.port[ADQ_PORT_PXIE].pin[ADQ_PIN_PXIE_STARC].invert_output = 0;
adq.port[ADQ_PORT_PXIE].pin[ADQ_PIN_PXIE_STARC].value = 1;
```

rather than

```
struct ADQParameters adq;
adq.port[9].pin[4].direction = ADQ_DIRECTION_OUT;
adq.port[9].pin[4].function = ADQ_FUNCTION_GPIO;
adq.port[9].pin[4].invert_output = 0;
adq.port[9].pin[4].value = 1;
```

Values

`ADQ_PIN_PXIE_TRIG0 (0)`
: A constant specifying the PXI TRIG0 pin.

`ADQ_PIN_PXIE_TRIG1 (1)`
: A constant specifying the PXI TRIG1 pin.

`ADQ_PIN_PXIE_STARA (2)`
: A constant specifying the PXIe STARA pin.

`ADQ_PIN_PXIE_STARB (3)`
: A constant specifying the PXIe STARB pin.

`ADQ_PIN_PXIE_STARC (4)`
: A constant specifying the PXIe STARC pin.

```
enum ADQImpedance {
    ADQ_IMPEDANCE_50_OHM  = 0,
    ADQ_IMPEDANCE_HIGH    = 1,
    ADQ_IMPEDANCE_100_OHM = 2
}
```

Description

An enumeration of the impedance values of the pins in the digitizer's ports. See Section 8 for details.

Values

`ADQ_IMPEDANCE_50_OHM (0)`

This constant specifies 50 Ohm.

`ADQ_IMPEDANCE_HIGH (1)`

This constant specifies a high impedance. The impedance value may differ between ports. Refer to the product datasheet [2] [3] [4] for typical values.

`ADQ_IMPEDANCE_100_OHM (2)`

This constant specifies 100 Ohm.

```
enum ADQDirection {
    ADQ_DIRECTION_IN    = 0,
    ADQ_DIRECTION_OUT   = 1,
    ADQ_DIRECTION_INOUT = 2
}
```

Description

An enumeration of the direction values of the pins in the digitizer's ports. See Section 8 for details.

Values

`ADQ_DIRECTION_IN (0)`

A constant specifying that the pin should be configured as an input.

`ADQ_DIRECTION_OUT (1)`

A constant specifying that the pin should be configured as an output.

`ADQ_DIRECTION_INOUT (2)`

A value used by the constant (read-only) parameter `direction` indicating that the corresponding pin is bidirectional.

```
enum ADQEdge {
    ADQ_EDGE_FALLING = 0,
    ADQ_EDGE_RISING  = 1,
    ADQ_EDGE_BOTH    = 2
}
```

Description

An enumeration of the edge selection for event sources. Not all event sources support edge selection, e.g. `ADQ_EVENT_SOURCE_SOFTWARE` only supports `ADQ_EDGE_RISING`.

Values

`ADQ_EDGE_FALLING (0)`

A constant specifying falling edge sensitivity.

`ADQ_EDGE_RISING (1)`

A constant specifying rising edge sensitivity.

`ADQ_EDGE_BOTH (2)`

A constant specifying sensitivity to both edges.

```
enum ADQPolarity {
    ADQ_POLARITY_INVALID  = 0,
    ADQ_POLARITY_NEGATIVE = 1,
    ADQ_POLARITY_POSITIVE = 2
}
```

Description

An enumeration for polarity. The values are used to e.g. indicate the pulse polarity in an application with unipolar pulse data. This is the target use case for the FWPD firmware and the associated PD signal processing module, see Section 5.7.

Values

`ADQ_POLARITY_INVALID (0)`

The invalid polarity. This constant is commonly used to signal the absence of a polarity, often implying that the function is disabled.

`ADQ_POLARITY_NEGATIVE (1)`

A constant specifying negative polarity.

`ADQ_POLARITY_POSITIVE (2)`

A constant specifying positive polarity.

```
enum ADQClockGenerator {
    ADQ_CLOCK_GENERATOR_INTERNAL_PLL    = 1,
    ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK = 2
}
```

Description

An enumeration of the digitizer's clock generation modes. See Section 4.1 for details.

Values

`ADQ_CLOCK_GENERATOR_INTERNAL_PLL (1)`

A constant specifying clock generation using the digitizer's internal PLL.

`ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK (2)`

A constant specifying external clock generation, with the clock supplied via the CLK port.

```
enum ADQReferenceClockSource {
    ADQ_REFERENCE_CLOCK_SOURCE_INTERNAL    = 1,
    ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK    = 2,
    ADQ_REFERENCE_CLOCK_SOURCE_PXIE_10M    = 3,
    ADQ_REFERENCE_CLOCK_SOURCE_MTCA_TCLKA = 4,
    ADQ_REFERENCE_CLOCK_SOURCE_MTCA_TCLKB = 5,
    ADQ_REFERENCE_CLOCK_SOURCE_PXIE_100M   = 6
}
```

Description

An enumeration of the digitizer's reference clock sources. Not all digitizers support every reference clock source. See Section 4.2 for details.

Values

`ADQ_REFERENCE_CLOCK_SOURCE_INTERNAL (1)`

A constant specifying the internal reference clock.

`ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK (2)`

A constant specifying an external reference clock supplied via the CLK port.

`ADQ_REFERENCE_CLOCK_SOURCE_PXIE_10M (3)`

A constant specifying the 10 MHz reference clock in a PXIe backplane. Only available for PXIe form factor digitizers.

`ADQ_REFERENCE_CLOCK_SOURCE_MTCA_TCLKA (4)`

A constant specifying the TCLKA reference clock in a MicroTCA backplane. Only available for MTCA form factor digitizers.

`ADQ_REFERENCE_CLOCK_SOURCE_MTCA_TCLKB (5)`

A constant specifying the TCLKB reference clock in a MicroTCA backplane. Only available for MTCA form factor digitizers.

`ADQ_REFERENCE_CLOCK_SOURCE_PXIE_100M (6)`

A constant specifying the 100 MHz reference clock in a PXIe backplane. Only available for PXIe form factor digitizers.

```
enum ADQFunction {
    ADQ_FUNCTION_INVALID                = 0,
    ADQ_FUNCTION_PATTERN_GENERATOR0     = 1,
    ADQ_FUNCTION_PATTERN_GENERATOR1     = 2,
    ADQ_FUNCTION_GPIO                   = 3,
    ADQ_FUNCTION_PULSE_GENERATOR0       = 4,
    ADQ_FUNCTION_PULSE_GENERATOR1       = 5,
    ADQ_FUNCTION_PULSE_GENERATOR2       = 6,
    ADQ_FUNCTION_PULSE_GENERATOR3       = 7,
    ADQ_FUNCTION_TIMESTAMP_SYNCHRONIZATION = 8,
    ADQ_FUNCTION_USER_LOGIC             = 9,
    ADQ_FUNCTION_DAISY_CHAIN            = 10,
    ADQ_FUNCTION_RECORD_STOP            = 11
}
```

Description

An enumeration of the digitizer's function modules. See Section 7 for details.

Values

`ADQ_FUNCTION_INVALID (0)`

A constant specifying an invalid function module. This value is commonly used to signal the absence of a function.

`ADQ_FUNCTION_PATTERN_GENERATOR0 (1)`

A constant specifying the first pattern generator module.

`ADQ_FUNCTION_PATTERN_GENERATOR1 (2)`

A constant specifying the second pattern generator module.

`ADQ_FUNCTION_GPIO (3)`

A constant specifying GPIO functionality.

`ADQ_FUNCTION_PULSE_GENERATOR0 (4)`

A constant specifying the first pattern generator module.

`ADQ_FUNCTION_PULSE_GENERATOR1 (5)`

A constant specifying the second pattern generator module.

`ADQ_FUNCTION_PULSE_GENERATOR2 (6)`

A constant specifying the third pattern generator module.

`ADQ_FUNCTION_PULSE_GENERATOR3 (7)`

A constant specifying the fourth pattern generator module.

`ADQ_FUNCTION_TIMESTAMP_SYNCHRONIZATION (8)`

A constant specifying the timestamp synchronization function.

`ADQ_FUNCTION_USER_LOGIC (9)`

A constant specifying that the pin is controlled from the user logic. For more information, refer to the development kit user guide [8]).

`ADQ_FUNCTION_DAISY_CHAIN (10)`

A constant specifying the daisy chain function.

`ADQ_FUNCTION_RECORD_STOP (11)`

A constant specifying a signal that is pulsed at the end of a record.

```
enum ADQPatternGeneratorOperation {
    ADQ_PATTERN_GENERATOR_OPERATION_TIMER = 0,
    ADQ_PATTERN_GENERATOR_OPERATION_EVENT = 1
}
```

Description

An enumeration of the operation specified in a pattern generator instruction (`op`). See Section 7.1 for details.

Values

`ADQ_PATTERN_GENERATOR_OPERATION_TIMER (0)`

A constant specifying the timer operation. See Section 7.1.1 for details.

`ADQ_PATTERN_GENERATOR_OPERATION_EVENT (1)`

A constant specifying the event operation. See Section 7.1.1 for details.

```
enum ADQMarkerMode {
    ADQ_MARKER_MODE_HOST_AUTO   = 0,
    ADQ_MARKER_MODE_HOST_MANUAL = 1,
    ADQ_MARKER_MODE_USER_ADDR   = 2
}
```

Description

An enumeration of the marker mode, i.e. how the data transfer process should handle the markers. See Section 10 for details.

Values

`ADQ_MARKER_MODE_HOST_AUTO (0)`

A constant specifying that the markers are handled automatically by the host computer, out of sight from the user application. This mode implies the use of `WaitForRecordBuffer()` and `ReturnRecordBuffer()` to read out data from the digitizer.

`ADQ_MARKER_MODE_HOST_MANUAL (1)`

A constant specifying that the markers are manually handled by the user. This mode implies the use of `WaitForP2pBuffers()` and `UnlockP2pBuffers()` to read out data from the digitizer.

`ADQ_MARKER_MODE_USER_ADDR (2)`

A constant specifying that the markers are transferred to the *user specified* `marker_buffer_bus_address`. This mode implies that the host computer RAM is not the target endpoint of the data transfer process. See Section 10 for more information.

```
enum ADQMemoryOwner {
    ADQ_MEMORY_OWNER_API  = 0,
    ADQ_MEMORY_OWNER_USER = 1
}
```

Description

An enumeration of the memory ownership modes used by the data readout parameter `memory_owner`.

Values

`ADQ_MEMORY_OWNER_API (0)`

A constant signifying that the memory of the data readout process is owned by the API.

`ADQ_MEMORY_OWNER_USER (1)`

A constant signifying that the memory of the data readout process is owned by the user. This value is currently unsupported on ADQ32 and ADQ33.

```
enum ADQTimestampSynchronizationMode {
    ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_DISABLE = 0,
    ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_FIRST   = 1,
    ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_ALL     = 2
}
```

Description

An enumeration of the timestamp synchronization modes.

Values

`ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_DISABLE (0)`

A constant specifying that the timestamp synchronization should be disabled.

`ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_FIRST (1)`

A constant specifying that the timestamp should synchronize on the first event.

`ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_ALL (2)`

A constant specifying that the timestamp should synchronize on every event.

```
enum ADQArm {
    ADQ_ARM_IMMEDIATELY          = 0,
    ADQ_ARM_AT_ACQUISITION_START = 1
}
```

Description

An enumeration specifying when the function should be armed.

Values

`ADQ_ARM_IMMEDIATELY (0)`

A constant specifying that the function should be armed in the call to `SetParameters()`.

`ADQ_ARM_AT_ACQUISITION_START (1)`

A constant specifying that the function should be armed in the call to `StartDataAcquisition()`.

```
enum ADQFirmwareType {
    ADQ_FIRMWARE_TYPE_INVALID = 0,
    ADQ_FIRMWARE_TYPE_FWDAQ   = 1,
    ADQ_FIRMWARE_TYPE_FWATD   = 2,
    ADQ_FIRMWARE_TYPE_FWPD    = 3
}
```

Description

An enumeration of the firmware types.

Values

`ADQ_FIRMWARE_TYPE_INVALID (0)`
: A constant specifying an invalid or unknown firmware type.

`ADQ_FIRMWARE_TYPE_FWDAQ (1)`
: A constant specifying a FWDAQ firmware.

`ADQ_FIRMWARE_TYPE_FWATD (2)`
: A constant specifying a FWATD firmware.

`ADQ_FIRMWARE_TYPE_FWPD (3)`
: A constant specifying a FWPD firmware.

```
enum ADQCommunicationInterface {
    ADQ_COMMUNICATION_INTERFACE_INVALID = 0,
    ADQ_COMMUNICATION_INTERFACE_PCIE    = 1,
    ADQ_COMMUNICATION_INTERFACE_USB     = 2
}
```

Description

An enumeration of the communication interfaces.

Values

`ADQ_COMMUNICATION_INTERFACE_INVALID (0)`
: A constant specifying an invalid or unknown communication interface.

`ADQ_COMMUNICATION_INTERFACE_PCIE (1)`
: A constant specifying a PCI-express communication interface.

`ADQ_COMMUNICATION_INTERFACE_USB (2)`
: A constant specifying a USB communication interface.

```
enum ADQCoefficientFormat {
    ADQ_COEFFICIENT_FORMAT_DOUBLE      = 0,
    ADQ_COEFFICIENT_FORMAT_FIXED_POINT = 1
}
```

Description

An enumeration of the coefficient formats that can be used when setting the coefficient values of the FIR filter (Section 5.4).

Values

`ADQ_COEFFICIENT_FORMAT_DOUBLE (0)`

A constant specifying double-precision floating point numbers as the coefficient format. When this format is used, values from the `coefficient` array are rounded and written to the filter.

`ADQ_COEFFICIENT_FORMAT_FIXED_POINT (1)`

A constant specifying fixed point numbers as the coefficient format. When this format is used, values from the `coefficient_fixed_point` array are rounded and written to the filter.

```
enum ADQRoundingMethod {
    ADQ_ROUNDING_METHOD_TIE_AWAY_FROM_ZERO = 0,
    ADQ_ROUNDING_METHOD_TIE_TOWARDS_ZERO   = 1,
    ADQ_ROUNDING_METHOD_TIE_TO_EVEN        = 2
}
```

Description

An enumeration of the rounding methods that can be used when setting the coefficient values of the FIR filter using the `ADQ_COEFFICIENT_FORMAT_DOUBLE` coefficient `format`. All rounding methods round to the nearest integer, but differ in the way tie breaks are handled.

Values

`ADQ_ROUNDING_METHOD_TIE_AWAY_FROM_ZERO (0)`

A constant specifying the rounding method where tie breaks are rounded away from zero, e.g. -10.5 is rounded to -11 and 10.5 is rounded to 11.

`ADQ_ROUNDING_METHOD_TIE_TOWARDS_ZERO (1)`

A constant specifying the rounding method where tie breaks are rounded towards zero, e.g. -10.5 is rounded to -10 and 10.5 is rounded to 10.

`ADQ_ROUNDING_METHOD_TIE_TO_EVEN (2)`

A constant specifying the rounding method where tie breaks are rounded to the nearest even integer, e.g. 10.5 is rounded to 10 and 11.5 is rounded to 12.

```
enum ADQUserLogic {
    ADQ_USER_LOGIC_RESERVED = 0,
    ADQ_USER_LOGIC1         = 1,
    ADQ_USER_LOGIC2         = 2
}
```

Description

An enumeration of the user logic areas.

Values

`ADQ_USER_LOGIC_RESERVED (0)`

Reserved.

`ADQ_USER_LOGIC1 (1)`

A constant specifying user logic area 1.

`ADQ_USER_LOGIC2 (2)`

A constant specifying user logic area 2.

```
enum ADQEeprom {
    ADQ_EEPROM_INVALID       = 0,
    ADQ_EEPROM_MOTHERBOARD   = 1,
    ADQ_EEPROM_DAUGHTERBOARD = 2,
    ADQ_EEPROM_USER          = 3
}
```

Description

An enumeration of the EEPROM areas.

Values

`ADQ_EEPROM_INVALID (0)`

Reserved.

`ADQ_EEPROM_MOTHERBOARD (1)`

A constant specifying the motherboard's EEPROM.

`ADQ_EEPROM_DAUGHTERBOARD (2)`

A constant specifying the daughterboard's EEPROM.

`ADQ_EEPROM_USER (3)`

A constant specifying the user's EEPROM area.

```
enum ADQHWIFEnum {
    HWIF_USB       = 0,
    HWIF_PCIE      = 1,
    HWIF_USB3      = 2,
    HWIF_PCIELITE  = 3,
    HWIF_ETH_ADQ7  = 4,
    HWIF_ETH_ADQ14 = 5,
    HWIF_QPCIE     = 7,
    HWIF_OTHER     = 8
}
```

Description

An enumeration of the hardware interfaces, used in `ADQControlUnit_ListDevices()`.

```
enum ADQProductID_Enum {
    PID_ADQ214         = 0x0001,
    PID_ADQ114         = 0x0003,
    PID_ADQ112         = 0x0005,
    PID_SphinxHS       = 0x000B,
    PID_SphinxLS       = 0x000C,
    PID_ADQ108         = 0x000E,
    PID_ADQDSP         = 0x000F,
    PID_SphinxAA14     = 0x0011,
    PID_SphinxAA16     = 0x0012,
    PID_ADQ412         = 0x0014,
    PID_ADQ212         = 0x0015,
    PID_SphinxAA_LS2   = 0x0016,
    PID_SphinxHS_LS2   = 0x0017,
    PID_SDR14          = 0x001B,
    PID_ADQ1600        = 0x001C,
    PID_SphinxXT       = 0x001D,
    PID_ADQ208         = 0x001E,
    PID_DSU            = 0x001F,
    PID_ADQ14          = 0x0020,
    PID_SDR14RF        = 0x0021,
    PID_EV12AS350_EVM = 0x0022,
    PID_ADQ7           = 0x0023,
    PID_ADQ8           = 0x0026,
    PID_ADQ12          = 0x0027,
    PID_ADQ32          = 0x0031,
    PID_ADQSM          = 0x0032,
    PID_ADQ36          = 0x0033,
    PID_ADQ30          = 0x0034,
    PID_TX320          = 0x201A,
    PID_RX320          = 0x201C,
    PID_S6000          = 0x2019
}
```

Description

An enumeration of the product IDs.

```
enum ADQAnalogInput {
    ADQ_ANALOG_INPUT_INVALID = 0,
    ADQ_ANALOG_INPUT_A       = 1,
    ADQ_ANALOG_INPUT_B       = 2,
    ADQ_ANALOG_INPUT_C       = 3,
    ADQ_ANALOG_INPUT_D       = 4,
    ADQ_ANALOG_INPUT_E       = 5,
    ADQ_ANALOG_INPUT_F       = 6,
    ADQ_ANALOG_INPUT_G       = 7,
    ADQ_ANALOG_INPUT_H       = 8
}
```

Description

An enumeration of the analog inputs of a digitizer.

Values

`ADQ_ANALOG_INPUT_INVALID (0)`
: A constant specifying an invalid analog input.

`ADQ_ANALOG_INPUT_A (1)`
: A constant specifying the analog input labeled A.

`ADQ_ANALOG_INPUT_B (2)`
: A constant specifying the analog input labeled B.

`ADQ_ANALOG_INPUT_C (3)`
: A constant specifying the analog input labeled C.

`ADQ_ANALOG_INPUT_D (4)`
: A constant specifying the analog input labeled D.

`ADQ_ANALOG_INPUT_E (5)`
: A constant specifying the analog input labeled E.

`ADQ_ANALOG_INPUT_F (6)`
: A constant specifying the analog input labeled F.

`ADQ_ANALOG_INPUT_G (7)`
: A constant specifying the analog input labeled G.

`ADQ_ANALOG_INPUT_H (8)`
: A constant specifying the analog input labeled H.

A.3 Structures

This section lists the data structures used when configuring and controlling the digitizer. These are defined in the ADQAPI header file `ADQAPI.h` and versioned by the two constants `ADQAPI_VERSION_MAJOR` and `ADQAPI_VERSION_MAJOR`. See `ADQAPI_ValidateVersion()` for more information about how to implement version validation in the user application space.

A.3.1 Initialization Parameters

This section lists the structures used to configure the digitizer during the initialization phase. See Section 15.4 for a description of the context in which these objects are used.

```
struct ADQClockSystemParameters {
    enum ADQParameterId           id;
    int32_t                       reserved;
    enum ADQClockGenerator        clock_generator;
    enum ADQReferenceClockSource  reference_source;
    double                        sampling_frequency;
    double                        reference_frequency;
    double                        delay_adjustment;
    int32_t                       low_jitter_mode_enabled;
    int32_t                       delay_adjustment_enabled;
    uint64_t                      magic;
}
```

Description

This struct defines the parameters of the digitizer's clock system and is used during the initialization phase (Section 15.4). See Section 4 for a high-level description of the clock system.

After initialization, the struct can be found as a member of the read-only `constant` parameters, where it holds the current clock system configuration.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_CLOCK_SYSTEM`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`clock_generator (enum ADQClockGenerator)`

An `ADQClockGenerator` that will be used to generate the sampling frequency clocks of the digitizer. The default value is `ADQ_CLOCK_GENERATOR_INTERNAL_PLL`. Valid values are:

- `ADQ_CLOCK_GENERATOR_INTERNAL_PLL`
- `ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK`

See Section 4.1 for a high-level description of the alternatives.

`reference_source (enum ADQReferenceClockSource)`

An `ADQReferenceClockSource` that will be used as reference clock for the internal PLL of the digitizer, assuming the `clock_generator` is set to `ADQ_CLOCK_GENERATOR_INTERNAL_PLL`. The default value is `ADQ_REFERENCE_CLOCK_SOURCE_INTERNAL`. Valid values are:

- `ADQ_REFERENCE_CLOCK_SOURCE_INTERNAL`
- `ADQ_REFERENCE_CLOCK_SOURCE_PORT_CLK`
- `ADQ_REFERENCE_CLOCK_SOURCE_PXIE_10M` (only ADQ36-PXIe)

See Section 4.2 for a high-level description of the alternatives.

`sampling_frequency (double)`

The desired sampling frequency, in units of Hz.

`reference_frequency (double)`

The supplied reference frequency, in units of Hz.

`delay_adjustment (double)`

The desired reference clock delay adjustment, in units of seconds. Requires that `delay_adjustment_enabled` is set to take effect.

`low_jitter_mode_enabled (int32_t)`

Enable or disable the low jitter mode of the internal PLL. See Section 4 for a high-level description of the low-jitter mode.

`delay_adjustment_enabled (int32_t)`

Enable or disable the reference clock delay adjustment. See Section 4 for a high-level description of the delay adjustment feature.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQInputRoutingParameters {
    enum ADQParameterId                     id;
    int32_t                                 reserved;
    struct ADQInputRoutingParametersChannel channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                magic;
}
```

Description

This struct defines the parameters that determine which analog input is connected to which digitizer channel.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_INPUT_ROUTING`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQInputRoutingParametersChannel)`

An array of `ADQInputRoutingParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQInputRoutingParametersChannel {
    enum ADQAnalogInput  input;
    int32_t              reserved;
}
```

Description

This struct is a member of `ADQInputRoutingParameters` and defines the input routing parameters for a channel.

Members

`input (enum ADQAnalogInput)`

The analog input connected to this digitizer channel. Channels that are not present in the digitizer firmware must have this field set to `ADQ_ANALOG_INPUT_INVALID`.

`reserved (int32_t)`

Reserved

A.3.2 Configuration Parameters

This section lists the structures used to configure the digitizer before initiating the acquisition process. See Section 15.5 for a description of the context in which these objects are used.

```
struct ADQParameters {
    enum ADQParameterId                 id;
    int32_t                             reserved;
    struct ADQConstantParameters        constant;
    struct ADQAnalogFrontendParameters  afe;
    struct ADQPortParameters            port[ADQ_MAX_NOF_PORTS];
    struct ADQEventSourceParameters     event_source;
    struct ADQFunctionParameters        function;
    struct ADQTestPatternParameters     test_pattern;
    struct ADQSignalProcessingParameters signal_processing;
    struct ADQDataAcquisitionParameters acquisition;
    struct ADQDataTransferParameters    transfer;
    struct ADQDataReadoutParameters     readout;
    uint64_t                            magic;
}
```

Description

This struct defines the entire parameter space of the digitizer. This means that each member struct is *also* an object that may interact with the configuration functions separately (see Section A.4.3).

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_TOP`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`constant (struct ADQConstantParameters)`

An `ADQConstantParameters` struct holding the constant parameters, i.e. parameters that cannot be modified. These include values such as the number of channels, each channel's base sampling rate, various labels and other useful properties of the digitizer.

`afe (struct ADQAnalogFrontendParameters)`

An `ADQAnalogFrontendParameters` struct holding the parameters of the analog front-end for all the channels of the digitizer. See Section 2 for a high-level description.

`port[ADQ_MAX_NOF_PORTS] (struct ADQPortParameters)`

An array of `ADQPortParameters` structs where each entry holds the parameters of a specific port. The array is intended to be indexed by using the enumeration `ADQPort`. See Section 8 for a

high-level description.

`event_source (struct ADQEventSourceParameters)`

An `ADQEventSourceParameters` struct holding the parameters of the digitizer's event sources. See Section 6 for a high-level description.

`function (struct ADQFunctionParameters)`

An `ADQFunctionParameters` struct holding the parameters of the digitizer's function modules. See Section 7 for a high-level description.

`test_pattern (struct ADQTestPatternParameters)`

An `ADQTestPatternParameters` struct holding the parameters of the digitizer's test pattern generator. See Section 11 for a high-level description.

`signal_processing (struct ADQSignalProcessingParameters)`

An `ADQSignalProcessingParameters` struct holding the parameters of the digitizer's signal processing modules. See Section 5 for a high-level description of these modules.

`acquisition (struct ADQDataAcquisitionParameters)`

An `ADQDataAcquisitionParameters` struct holding the parameters of the data acquisition process. See Section 9 for a high-level description.

`transfer (struct ADQDataTransferParameters)`

An `ADQDataTransferParameters` struct holding the parameters of the data transfer process. See Section 10 for a high-level description.

`readout (struct ADQDataReadoutParameters)`

An `ADQDataReadoutParameters` struct holding the parameters of the data readout process. See Section 10 for a high-level description.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQAnalogFrontendParameters {
    enum ADQParameterId                         id;
    int32_t                                     reserved;
    struct ADQAnalogFrontendParametersChannel   channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                    magic;
}
```

Description

This struct defines the parameters of the analog front-end for all channels of the digitizer. See Section 2 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_ANALOG_FRONTEND`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQAnalogFrontendParametersChannel)`

An array of `ADQAnalogFrontendParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQAnalogFrontendParametersChannel {
    double  input_range;
    double  dc_offset;
}
```

Description

This struct is a member of `ADQAnalogFrontendParameters` and defines analog front-end parameters for a channel.

Members

`input_range (double)`

The channel's input range in millivolts. The default value depends on the digitizer model.

`dc_offset (double)`

The channel's analog DC offset in millivolts. The default value is zero.

```
struct ADQAtdParameters {
    enum ADQParameterId             id;
    int32_t                         reserved;
    struct ADQAtdParametersCommon   common;
    struct ADQAtdParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                        magic;
}
```

Description

This struct defines the parameters of the ATD signal processing module for all channels of the digitizer. See Section 5.6 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_ATD`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`common (struct ADQAtdParametersCommon)`

A `ADQAtdParametersCommon` struct holding the ATD signal processing module parameters that apply to all channels.

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQAtdParametersChannel)`

An array of `ADQAtdParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_transfer_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQAtdParametersCommon {
    int64_t           nof_accumulations;
    enum ADQFunction  accumulation_grid_synchronization_source;
    int32_t           reserved;
}
```

Description

This struct is a member of `ADQAtdParameters` and defines the ATD signal processing module parameters for all channels.

Members

`nof_accumulations (int64_t)`

Number of records to accumulate before outputting the accumulated result as a record to the user.

`accumulation_grid_synchronization_source (enum ADQFunction)`

The accumulation grid synchronization source. See Section 5.6.5 for a high-level description. Valid values are:

- `ADQ_FUNCTION_INVALID`
- `ADQ_FUNCTION_PATTERN_GENERATOR0`
- `ADQ_FUNCTION_PATTERN_GENERATOR1`

The default value is `ADQ_FUNCTION_INVALID` which implies that the grid synchronization mechanism is not active.

`reserved (int32_t)`

Reserved

```
struct ADQAtdParametersChannel {
    struct ADQAtdParametersChannelThresholdFilter  threshold_filter;
}
```

Description

This struct is a member of `ADQAtdParameters` and defines the ATD signal processing module parameters for a channel.

Members

`threshold_filter (struct ADQAtdParametersChannelThresholdFilter)`

A `ADQAtdParametersChannelThresholdFilter` struct holding the threshold filter parameters for the channel.

```
struct ADQAtdParametersChannelThresholdFilter {
    int32_t                     enabled;
    enum ADQPolarity            polarity;
    enum ADQRoundingMethod      rounding_method;
    enum ADQCoefficientFormat   format;
    int64_t                     level;
    int64_t                     baseline;
    double                      coefficient[ADQ_MAX_NOF_ATD_THRESHOLD_FILTER_COEFFICIENTS];
    int32_t                     coefficient_fixed_point[ADQ_MAX_NOF_ATD_THRESHOLD_FILTER_COEFFICIENTS]
    int32_t                     reserved;
}
```

Description

This struct is a member of `ADQAtdParameters` and defines threshold filter parameters for the ATD signal processing module.

Members

`enabled (int32_t)`

Set this parameter to a nonzero value to enable the ATD signal processing module threshold filter (Section 5.6.3.) Depending on the `polarity`, samples above or below the threshold `level` will be set to the `baseline` value.

`polarity (enum ADQPolarity)`

Threshold polarity for the ATD signal processing module threshold filter. With `ADQ_POLARITY_NEGATIVE`, samples with values below the `level` will pass through unchanged, while samples with values over the `level` are set to the `baseline` value. With `ADQ_POLARITY_POSITIVE`, samples with values above the `level` will pass through unchanged, while samples with values under the `level` are set to the `baseline` value.

`rounding_method (enum ADQRoundingMethod)` Write-only

The rounding method that is used to convert the floating point values in the `coefficient` array to the fixed point precision of the filter. The default value is `ADQ_ROUNDING_METHOD_TIE_AWAY_FROM_ZERO`. This parameter is write-only.

`format (enum ADQCoefficientFormat)` Write-only

The coefficient format to use when setting the threshold filter coefficients. Refer to the enumeration `ADQCoefficientFormat` for more information. The default value is `ADQ_COEFFICIENT_FORMAT_DOUBLE`. This parameter is write-only.

`level (int64_t)`

Threshold level for the ATD signal processing module threshold filter. The threshold level is specified as ADC codes with valid range of $[-2^{15}, 2^{15} - 1]$

`baseline (int64_t)`

Baseline level for the ATD signal processing module threshold filter. The baseline level is specified

as ADC codes with valid range of $[-2^{15}, 2^{15} - 1]$

`coefficient[ADQ_MAX_NOF_ATD_THRESHOLD_FILTER_COEFFICIENTS]` (`double`)

The threshold filter coefficients, in double-precision floating point format. When setting the parameters of the filter, this array is only used if the `format` is set to `ADQ_COEFFICIENT_FORMAT_DOUBLE`. The default values are set so the filter acts as a bypass.

`coefficient_fixed_point[ADQ_MAX_NOF_ATD_THRESHOLD_FILTER_COEFFICIENTS]` (`int32_t`)

The threshold filter coefficients, in fixed point format. When setting the parameters of the filter, this array is only used if the `format` is set to `ADQ_COEFFICIENT_FORMAT_FIXED_POINT`. The default values are set so the filter acts as a bypass.

`reserved` (`int32_t`)

Reserved

```
struct ADQConstantParameters {
    enum ADQParameterId                                 id;
    int32_t                                             nof_channels;
    int32_t                                             nof_acquisition_channels;
    int32_t                                             nof_transfer_channels;
    int32_t                                             nof_pattern_generators;
    int32_t                                             max_nof_pattern_generator_instructions;
    int32_t                                             nof_pulse_generators;
    int32_t                                             nof_matrix_inputs;
    char                                                dna[40];
    char                                                serial_number[16];
    char                                                product_name[32];
    char                                                product_options[32];
    struct ADQConstantParametersFirmware                firmware;
    struct ADQClockSystemParameters                     clock_system;
    struct ADQConstantParametersChannel                 channel[ADQ_MAX_NOF_CHANNELS];
    struct ADQConstantParametersPort                    port[ADQ_MAX_NOF_PORTS];
    struct ADQConstantParametersCommunicationInterface  communication_interface;
    uint64_t                                            eeprom_user_area_size;
    uint64_t                                            dram_size;
    int32_t                                             record_buffer_size_step;
    uint64_t                                            magic;
}
```

Description

This struct defines the constant parameters of the digitizer, i.e. parameters that cannot be modified by the user. It offers a way to programatically query information about the digitizer.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_CONSTANT`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`nof_channels (int32_t)`

The number of physical channels.

`nof_acquisition_channels (int32_t)`

The number of acquisition channels.

`nof_transfer_channels (int32_t)`

The number of transfer channels.

`nof_pattern_generators (int32_t)`

The number of pattern generators. See Section 7.1 for details.

`max_nof_pattern_generator_instructions (int32_t)`

The maximum number of pattern generator instructions. See Section 7.1 for details.

`nof_pulse_generators (int32_t)`

The number of pulse generators. See Section 7.2 for details.

`nof_matrix_inputs (int32_t)`

The number of inputs to the matrix event source. See Section 6.10 for details.

`dna[40] (char)`

The digitizer's *DNA* as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"0x000000001234ABCD"`. This value is a unique identifier that may be requested by TSPD support staff.

`serial_number[16] (char)`

The serial number as a zero-terminated array of ASCII characters, i.e. a C-string. This value is normally on the form `"SPD-09999"`.

`product_name[32] (char)`

The product name as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"ADQ32"`.

`product_options[32] (char)`

The product options as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"-DC-S2G5-BW1G0-R0V5-PCIe"`. The meaning of each option is described in the product datasheet. [2] [3] [4]

`firmware (struct ADQConstantParametersFirmware)`

This struct is a member of `ADQConstantParameters` and defines constant parameters for the firmware currently running on the digitizer.

`clock_system (struct ADQClockSystemParameters)`

The parameter set of the currently active clock system configuration.

Important

The clock system configuration cannot be modified through these parameters. See Section 15.4 for more information.

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQConstantParametersChannel)`

An array of `ADQConstantParametersChannel` structs where each element represents the constant parameters for a channel. The struct at index 0 targets the first channel.

`port[ADQ_MAX_NOF_PORTS] (struct ADQConstantParametersPort)`

An array of `ADQConstantParametersPort` structs where each element represents the constant parameters for a port. The array is intended to be indexed by using the enumeration `ADQPort`.

`communication_interface (struct ADQConstantParametersCommunicationInterface)`

This struct is a member of `ADQConstantParameters` and defines constant parameters for the communication interface between the host system and the digitizer.

`eeprom_user_area_size (uint64_t)`

The size (in bytes) of the area in the digitizer's nonvolatile memory that is dedicated to user data. Refer to Section 14 for details.

`dram_size (uint64_t)`

The total size, i.e. capacity, of the digitizer's on-board DRAM in bytes.

`record_buffer_size_step (int32_t)`

The required granularity of the `record_buffer_size` when the digitizer is configured

- for records with infinite length,
- for records with dynamic length (Section 9.1); or
- to continue on overflow (Section 10.6.3).

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQConstantParametersChannel {
    double                                  input_range[ADQ_MAX_NOF_INPUT_RANGES];
    char                                    label[8];
    int32_t                                 nof_adc_cores;
    int32_t                                 nof_input_ranges;
    int32_t                                 has_variable_dc_offset;
    int32_t                                 has_variable_input_range;
    int64_t                                 code_normalization;
    struct ADQConstantParametersFirFilter   fir_filter;
    struct ADQConstantParametersPdrx        pdrx;
    struct ADQConstantParametersAtd         atd;
    struct ADQConstantParametersPd          pd;
}
```

Description

This struct is a member of `ADQConstantParameters` and defines constant parameters for a channel.

Members

`input_range[ADQ_MAX_NOF_INPUT_RANGES]` (`double`)

An array of input ranges in millivolt. The number of valid entries is given by the value of `nof_input_ranges`.

`label[8]` (`char`)

The channel label as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"A"`, `"B"`, `"C"` etc. This label corresponds to the identifier printed on the digitizer's front panel.

`nof_adc_cores` (`int32_t`)

The number of ADC cores used to create the channel's data stream. This value is greater than one when several ADC cores are *interleaved* to achieve a higher effective sampling rate while running each core at a lower sampling rate.

`nof_input_ranges` (`int32_t`)

The number of valid input range entries in the array `input_range`.

`has_variable_dc_offset` (`int32_t`)

A boolean value where a nonzero value indicates that the channel supports a variable DC offset via the analog front-end parameter `dc_offset`.

`has_variable_input_range` (`int32_t`)

A boolean value where a nonzero value indicates that the channel supports a variable input range via the analog front-end parameter `input_range`.

`code_normalization` (`int64_t`)

A normalization factor that is used together with `input_range` to convert ADC codes to an input voltage.

`fir_filter` (`struct ADQConstantParametersFirFilter`)

This struct defines the constant parameters for the FIR filter of the channel.

`pdrx` (`struct ADQConstantParametersPdrx`)

This struct defines the constant parameters for the channel's PDRX module.

`atd` (`struct ADQConstantParametersAtd`)

This struct defines the constant parameters for the channel's ATD module.

`pd` (`struct ADQConstantParametersPd`)

This struct defines the constant parameters for the channel's PD module.

```
struct ADQConstantParametersPort {
    int32_t                         nof_pins;
    int32_t                         is_present;
    char                            label[16];
    struct ADQConstantParametersPin pin[ADQ_MAX_NOF_PINS];
}
```

Description

This struct is a member of `ADQConstantParameters` and defines constant parameters for a port (Section 8).

Members

`nof_pins` (`int32_t`)

The number of pins in the port. This value specifies the number of valid entries in the array `pin`.

`is_present` (`int32_t`)

A boolean value where a nonzero value indicates that the port is present. This is the same as testing if `nof_pins` is greater than zero.

`label[16]` (`char`)

The port label as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"TRIG"`, `"SYNC"`, `"CLK"`. This label corresponds to the identifier printed on the digitizer's front panel.

`pin[ADQ_MAX_NOF_PINS]` (`struct ADQConstantParametersPin`)

An array of `ADQConstantParametersPin` structs where each entry represents the constant parameters for a pin in the port.

```
struct ADQConstantParametersPin {
    enum ADQEventSource  event_source;
    enum ADQDirection    direction;
    int32_t              has_configurable_threshold;
    int32_t              reserved;
}
```

Description

This struct is a member of `ADQConstantParametersPort` and defines constant parameters for a pin in a port (Section 8).

Members

`event_source (enum ADQEventSource)`

The event source (Section 6) associated with the pin. The value is set to `ADQ_EVENT_SOURCE_INVALID` if events cannot be generated by the pin.

`direction (enum ADQDirection)`

The directionality of the pin. A pin with both input and output capabilities is reported with the value `ADQ_DIRECTION_INOUT`.

`has_configurable_threshold (int32_t)`

Pins with an associated event source (`event_source` != `ADQ_EVENT_SOURCE_INVALID`) may support a configurable threshold via the port event source parameter `threshold`. If this is supported by the pin, this value is nonzero.

`reserved (int32_t)`

Reserved

```
struct ADQConstantParametersFirmware {
    enum ADQFirmwareType  type;
    int32_t               reserved;
    char                  name[32];
    char                  revision[32];
    char                  customization[16];
    char                  part_number[16];
}
```

Description

This struct is a member of `ADQConstantParameters` and defines constant parameters for the firmware currently running on the digitizer.

Members

`type (enum ADQFirmwareType)`

The current firmware type expressed as a value from the enumeration `ADQFirmwareType`.

`reserved (int32_t)`

Reserved

`name[32] (char)`

The firmware name as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"1CH-FWDAQ-PCIE"`.

`revision[32] (char)`

The firmware revision as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"59000"`.

`customization[16] (char)`

The firmware customization as a zero-terminated array of ASCII characters, i.e. a C-string. This string designates whether the firmware is a standard firmware (`"STANDARD"`) or built from a development kit (`"DEVKIT"`).

`part_number[16] (char)`

The firmware part number as a zero-teminated array of ASCII characters, i.e. a C-string. This string is an identifier used by TSPD and is of limited use to the user.

```
struct ADQConstantParametersCommunicationInterface {
    enum ADQCommunicationInterface  type;
    int32_t                         link_width;
    int32_t                         link_generation;
    int32_t                         reserved;
}
```

Description

This struct is a member of `ADQConstantParameters` and defines constant parameters for the communication interface between the host system and the digitizer.

Members

`type (enum ADQCommunicationInterface)`

The type of communication interface between the host system and the digitizer.

`link_width (int32_t)`

For PCI-express, this value corresponds to the PCI-express link width, or number of active lanes. For non-PCIe communication interfaces, it is undefined.

`link_generation (int32_t)`

For PCI-express, this value corresponds to the PCI-express generation. For non-PCIe communication interfaces, it is undefined.

`reserved (int32_t)`

Reserved

```
struct ADQConstantParametersFirFilter {
    int32_t  is_present;
    int32_t  order;
    int32_t  nof_coefficients;
    int32_t  coefficient_bits;
    int32_t  coefficient_fractional_bits;
    int32_t  reserved;
}
```

Description

This struct is a member of `ADQConstantParametersChannel` and defines constant parameters for the FIR filter (Section 5.4).

Members

`is_present` (`int32_t`)
: A boolean value where a nonzero value indicates that the FIR filter is present in firmware.

`order` (`int32_t`)
: The FIR filter order.

`nof_coefficients` (`int32_t`)
: The number of programmable filter coefficients.

`coefficient_bits` (`int32_t`)
: The total number of bits in the fixed point representation of each coefficient.

`coefficient_fractional_bits` (`int32_t`)
: The number of fractional bits in the fixed point representation of each coefficient.

`reserved` (`int32_t`)
: Reserved

```
struct ADQConstantParametersPdrx {
    int32_t  is_present;
    int32_t  high_gain_channel;
    int32_t  equalizer_order;
    int32_t  nof_equalizer_coefficients;
    int32_t  reflection_filter_order;
    int32_t  nof_reflection_filter_coefficients;
}
```

Description

This struct is a member of `ADQConstantParametersChannel` and defines constant parameters for the channel's pulse detection range extension (PDRX) module (Section 5.5).

Members

`is_present` (`int32_t`)

A boolean value where a nonzero value indicates that the channel supports PDRX.

`high_gain_channel` (`int32_t`)

For channels supporting PDRX (`is_present` is nonzero), this value holds the index of the corresponding high-gain channel.

`equalizer_order` (`int32_t`)

The equalizer order.

`nof_equalizer_coefficients` (`int32_t`)

The number of programmable equalizer coefficients.

`reflection_filter_order` (`int32_t`)

The reflection filter order.

`nof_reflection_filter_coefficients` (`int32_t`)

The number of reflection filter coefficients.

```
struct ADQConstantParametersAtd {
    struct ADQConstantParametersAtdThresholdFilter  threshold_filter;
}
```

Description

This struct is a member of `ADQConstantParametersChannel` and defines constant parameters for the channel's ATD signal processing module (Section 5.6).

Members

`threshold_filter` (`struct ADQConstantParametersAtdThresholdFilter`)

This struct contains parameters for the ATD signal processing module threshold filter.

```
struct ADQConstantParametersAtdThresholdFilter {
    int64_t  nof_coefficients;
    int64_t  coefficient_bits;
    int64_t  coefficient_fractional_bits;
}
```

Description

This struct is a member of `ADQConstantParametersAtd` and defines constant parameters for the threshold filter of the channel's ATD signal processing module (Section 5.6).

Members

`nof_coefficients (int64_t)`

The number of coefficients for the threshold filter of the ATD signal processing module.

`coefficient_bits (int64_t)`

The number of bits used in the coefficient format for the threshold filter of the ATD signal processing module.

`coefficient_fractional_bits (int64_t)`

The number of fractional bits used in the coefficient format for the threshold filter of the ATD signal processing module.

```
struct ADQConstantParametersPd {
    int32_t  source_channel;
}
```

Description

This struct is a member of `ADQConstantParametersChannel` and defines constant parameters for the channel's PD signal processing module (Section 5.7).

Members

`source_channel (int32_t)`

For pulse attribute channels this value holds the index of the corresponding source channel. The value is only valid for FWPD.

```
struct ADQDaisyChainParameters {
    enum ADQParameterId  id;
    enum ADQEventSource  source;
    enum ADQEdge         edge;
    enum ADQArm          arm;
    int32_t              resynchronization_enabled;
    int32_t              position;
    uint64_t             magic;
}
```

Description

This struct defines the parameters for the daisy chain function. See Section 7.4 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DAISY_`

CHAIN. This is guaranteed if InitializeParameters() is called to initialize the parameter set.

source (enum ADQEventSource)

The event source used for the daisy chain. Valid values are:

- ADQ_EVENT_SOURCE_SOFTWARE
- ADQ_EVENT_SOURCE_TRIG
- ADQ_EVENT_SOURCE_SYNC
- ADQ_EVENT_SOURCE_GPIOA0
- ADQ_EVENT_SOURCE_GPIOB0
- ADQ_EVENT_SOURCE_PXIE_STARB
- ADQ_EVENT_SOURCE_PXIE_TRIG0
- ADQ_EVENT_SOURCE_PXIE_TRIG1
- ADQ_EVENT_SOURCE_LEVEL_MATRIX

The default value is ADQ_EVENT_SOURCE_INVALID.

edge (enum ADQEdge)

An ADQEdge which specifies the edge sensitivity of the source. The default value is ADQ_EDGE_RISING.

arm (enum ADQArm)

Specifies when the daisy chain function should be armed and ready to react to events from the selected event source. Valid values are:

- ADQ_ARM_IMMEDIATELY
- ADQ_ARM_AT_ACQUISITION_START

The default value is ADQ_ARM_IMMEDIATELY.

resynchronization_enabled (int32_t)

If set to 1, the daisy chain input signal propagates to the output on the next edge of the reference clock. If set to 0, the input signal is propagated as soon as possible. In practice, there will be a nonzero propagation delay since the signal still passes through analog buffers and decision logic on its way to the output. The default value is 1.

position (int32_t)

Specifies the position of the digitizer in the daisy chain. The position starts at 0.

magic (uint64_t)

A magic number to indicate the end of the parameter struct. This value should always be set to ADQ_PARAMETERS_MAGIC. This is guaranteed if InitializeParameters() is called to initialize the parameter set.

```
struct ADQDataAcquisitionParameters {
    enum ADQParameterId                          id;
    int32_t                                      reserved;
    struct ADQDataAcquisitionParametersCommon    common;
    struct ADQDataAcquisitionParametersChannel   channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                     magic;
}
```

Description

This struct defines the parameters for the data acquisition process. See Section 9 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DATA_ACQUISITION`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`common (struct ADQDataAcquisitionParametersCommon)`

A `ADQDataAcquisitionParametersCommon` struct holding parameters that apply to all channels.

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQDataAcquisitionParametersChannel)`

An array of `ADQDataAcquisitionParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_acquisition_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQDataAcquisitionParametersCommon {
    int64_t  reserved;
}
```

Description

This struct is a member of `ADQDataAcquisitionParameters` and defines data acquisition parameters that apply to all channels.

Members

`reserved (int64_t)`

Reserved

```
struct ADQDataAcquisitionParametersChannel {
    int64_t              horizontal_offset;
    int64_t              record_length;
    int64_t              nof_records;
    int64_t              bytes_per_sample;
    int64_t              dsu_forced_metadata_interval;
    int64_t              rearm_length;
    int64_t              dynamic_record_length_max;
    int64_t              dynamic_leading_edge_window_length;
    int64_t              dynamic_trailing_edge_window_length;
    enum ADQEventSource  trigger_source;
    enum ADQEdge         trigger_edge;
    enum ADQFunction     trigger_blocking_source;
    int32_t              zero_length_records_enabled;
    int32_t              dynamic_record_length_enabled;
    int32_t              reserved;
}
```

Description

This struct is a member of `ADQDataAcquisitionParameters` and defines data acquisition parameters for a channel.

Members

`horizontal_offset (int64_t)`

The horizontal offset for a record in samples, i.e. the offset between the trigger event and the first sample in the acquired record. A negative value captures data *before* the trigger event (pretrigger) and a strictly positive value delays the capture. The default value is 0. The valid range and step size depends on the digitizer model and its current firmware:

- `ADQ30`
  - `1CH-FWDAQ`, `1CH-FWATD`: valid range of $[-16360, 2^{35} - 8]$, step size of 8
- `ADQ32, ADQ33`
  - `2CH-FWDAQ`, `2CH-FWATD`: valid range of $[-16360, 2^{35} - 8]$, step size of 8
  - `1CH-FWDAQ`, `1CH-FWATD`: valid range of $[-16336, 2^{36} - 16]$, step size of 16
- `ADQ36`
  - `4CH-FWDAQ`: valid range of $[-16360, 2^{35} - 8]$, step size of 8
  - `2CH-FWDAQ`: valid range of $[-16336, 2^{36} - 16]$, step size of 16

Note

If `dynamic_record_length_enabled` is set to 1, the lower bound for the `horizontal_offset` will depend on the value of `dynamic_leading_edge_window_length`. Refer to the documentation for that parameter for more information.

`record_length (int64_t)`

The record length in samples. The value `ADQ_INFINITE_RECORD_LENGTH` may be used to indicate a record with infinite length, i.e. a record that never ends. This configuration has to be matched by a similar configuration of the data transfer and data readout processes. Refer to Section 10.5.7 for more details. The default value is 0, and the valid range is $[2, 2^{32} - 1]$. The `FWATD` firmware puts additional limitations on the record length, which are described in Section 5.6.4. Validation is only performed for active channels, see `nof_records`.

`nof_records (int64_t)`

The number of records to acquire. The value `ADQ_INFINITE_NOF_RECORDS` may be used to indicate an unbounded acquisition. A channel is disabled by setting this parameter to zero (the default value). The valid range is $[0, 2^{32} - 1]$.

`bytes_per_sample (int64_t)`

The number of bytes required to store a single sample. This parameter can currently not be changed from its default value.

`dsu_forced_metadata_interval (int64_t)`

Reserved

`rearm_length (int64_t)`

The rearm time in samples, i.e. the interval following the end of a record during which any trigger event is *ignored*. Refer to Section 9.2 for additional information.

- `ADQ30`
  - `1CH-FWDAQ, 1CH-FWATD`: valid range of $[16, 2^{32} - 1]$ in steps of 16 samples.
- `ADQ32, ADQ33`
  - `2CH-FWDAQ, 2CH-FWATD`: valid range of $[16, 2^{32} - 1]$ in steps of 16 samples.
  - `1CH-FWDAQ, 1CH-FWDAQ`: valid range of $[32, 2^{32} - 1]$ in steps of 32 samples.
- `ADQ36`
  - `4CH-FWDAQ, 4CH-FWATD`: valid range of $[16, 2^{32} - 1]$ in steps of 16 samples.
  - `2CH-FWDAQ, 2CH-FWDAQ`: valid range of $[32, 2^{32} - 1]$ in steps of 32 samples.

Important

The minimum value can change if the digitizer firmware is updated. However, the minimum value will at most correspond to the rearm time specified in the digitizer's datasheet. [1] [2] [3] [4]

`dynamic_record_length_max` (`int64_t`)

The maximum record length (in samples) for a record with dynamic length. The value `ADQ_INFINITE_RECORD_LENGTH` may be used to allow a record with infinite length, i.e. a record that never ends. The default value is `ADQ_INFINITE_RECORD_LENGTH`. In addition to this value, the valid range is $[2, 2^{32} - 1]$.

Important

This parameter is only used when `dynamic_record_length_enabled` is set to 1, and is ignore otherwise.

`dynamic_leading_edge_window_length` (`int64_t`)

The leading edge window length (in samples) for a record with dynamic length. The maximum length depends on the `horizontal_offset`, and must fulfill

$$\texttt{dynamic\_leading\_edge\_window\_length} \leq \{16360, 16336\} - |\min(\texttt{horizontal\_offset}, 0)|$$

where the constant $\{16360, 16336\}$ is the absolute value of the lower bound of the `horizontal_offset`, and depends on the digitizer model and its current firmware. The default value is 0.

Example

Let the `horizontal_offset` be -7880 for an `ADQ32-1CH` digitizer running `FWDAQ`. The maximum leading edge window length is calculated as

$$16336 - |-7880| = 8456.$$

Important

This parameter is only used when `dynamic_record_length_enabled` is set to 1, and must be set to 0 otherwise.

`dynamic_trailing_edge_window_length` (`int64_t`)

The trailing edge window length (in samples) for a record with dynamic length. The default value is 0. The valid range is $[2, 2^{32} - 1]$ when dynamic record length is enabled, and must be set to 0 otherwise.

`trigger_source` (`enum ADQEventSource`)

An `ADQEventSource` whose events are used to trigger a record. The default value is `ADQ_EVENT_SOURCE_SOFTWARE`. Not every event source can be used as a trigger source. Valid values are:

- `ADQ_EVENT_SOURCE_SOFTWARE`
- `ADQ_EVENT_SOURCE_TRIG`
- `ADQ_EVENT_SOURCE_LEVEL`
- `ADQ_EVENT_SOURCE_PERIODIC`
- `ADQ_EVENT_SOURCE_PXIE_STARB` (only ADQ36-PXIe)

- `ADQ_EVENT_SOURCE_SYNC`
- `ADQ_EVENT_SOURCE_GPIOA0`
- `ADQ_EVENT_SOURCE_GPIOB0` (only ADQ36-PXIe)
- `ADQ_EVENT_SOURCE_PXIE_TRIG0` (only ADQ36-PXIe)
- `ADQ_EVENT_SOURCE_PXIE_TRIG1` (only ADQ36-PXIe)
- `ADQ_EVENT_SOURCE_REFERENCE_CLOCK`
- `ADQ_EVENT_SOURCE_MATRIX`
- `ADQ_EVENT_SOURCE_LEVEL_MATRIX`
- `ADQ_EVENT_SOURCE_LEVEL_CHANNEL0`, `ADQ_EVENT_SOURCE_LEVEL_CHANNEL1`, etc. up to the index of the last channel of the digitizer.

Refer to the documentation for the respective enumeration value to understand the event source details.

`trigger_edge (enum ADQEdge)`

An `ADQEdge` which specifies the edge selection of the `trigger_source`. The default value is `ADQ_EDGE_RISING`.

`trigger_blocking_source (enum ADQFunction)`

An `ADQFunction` which specifies the trigger blocking source. See Section 9.5 for a high-level description. Valid values are:

- `ADQ_FUNCTION_INVALID`
- `ADQ_FUNCTION_PATTERN_GENERATOR0`
- `ADQ_FUNCTION_PATTERN_GENERATOR1`

The default value is `ADQ_FUNCTION_INVALID` which implies that the blocking mechanism is not active.

`zero_length_records_enabled (int32_t)`

Set to a nonzero value to enable the generation of a zero length record each time the trigger blocking mechanism transitions from the accept state to the block state without having observed any trigger events. Refer to Section 9.5.1 for additional information. The default value is zero (disabled).

Note

A zero length record only consists of a `header` so the data transfer process *must* enable propagation of metadata for a zero length record to propagate.

`dynamic_record_length_enabled (int32_t)`

Set to a nonzero value to enable the *acquisition* of records with dynamic length. The default value is zero (disabled). When enabled, the data transfer parameter `dynamic_record_length_enabled` must be set to `1`.

Important

Dynamic record length is not supported by FWATD.

`reserved (int32_t)`

Reserved

```
struct ADQDataTransferParameters {
    enum ADQParameterId                         id;
    int32_t                                     reserved;
    struct ADQDataTransferParametersCommon      common;
    struct ADQDataTransferParametersChannel     channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                    magic;
}
```

Description

This struct defines the parameters for the data transfer process (Section 10).

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DATA_TRANSFER`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`common (struct ADQDataTransferParametersCommon)`

A `ADQDataTransferParametersCommon` struct holding data transfer parameters that apply to all channels.

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQDataTransferParametersChannel)`

An array of `ADQDataTransferParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_transfer_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQDataTransferParametersCommon {
    int64_t             record_buffer_packed_size;
    int64_t             metadata_buffer_packed_size;
    double              overflow_hysteresis;
    enum ADQMarkerMode  marker_mode;
    int32_t             write_lock_enabled;
    int32_t             transfer_records_to_host_enabled;
    int32_t             packed_buffers_enabled;
    uint32_t            dsu_doorbell_value_mask;
    int32_t             dsu_operation_size;
}
```

Description

This struct is a member of `ADQDataTransferParameters` and defines data transfer parameters that apply to all channels.

Members

`record_buffer_packed_size (int64_t)`

The effective size of a packed record transfer buffer. The packed buffer contains the record transfer buffers for all active channels back-to-back. Each record transfer buffer has size `record_buffer_size`. The packed size is calculated by API. Since the buffer allocation always is page aligned, the allocated buffer size may be larger than the effective size. This parameter is only used when `packed_buffers_enabled` is set to 1.

`metadata_buffer_packed_size (int64_t)`

Like `record_buffer_packed_size` but for the metadata transfer buffers.

`overflow_hysteresis (double)`

The overflow hysteresis as a percentage of the total `dram_size`. This value only affects the data transfer process when `continue_on_overflow_enabled` is set to 1. The default value is 3% and signifies that following an overflow (Section 10.6.1) no new data is written until 3% of the capacity has been made available, i.e. the fill level has been reduced to 97%.

The value is subjected to rounding and is not intended to be controlled with high precision. The value read in a call to `GetParameters()` may be different from the requested hysteresis but reflects the one used by the digitizer. The valid range is [0, 100]. Refer to Section 10.6.3 for information.

`marker_mode (enum ADQMarkerMode)`

An `ADQMarkerMode` that specifies the way filled transfer buffers are detected. In most use cases the markers are handled by the API and no direct user interaction with markers is needed. For a high-level description see Section 10.2. There are three marker modes:

`ADQ_MARKER_MODE_HOST_AUTO`

The default value `ADQ_MARKER_MODE_HOST_AUTO` is used together with the data readout

interface (Section 10.5). In this case, markers are handled by the API, out of sight from the user. Records are received by the user application by calling `WaitForRecordBuffer()`.

`ADQ_MARKER_MODE_HOST_MANUAL`

The value `ADQ_MARKER_MODE_HOST_MANUAL` is used together with the data transfer interface (Section 10.4). The markers will be transferred to memory owned by the API in the host computer's RAM and the user application will use `WaitForP2pBuffers()` to detect filled transfer buffers.

`ADQ_MARKER_MODE_USER_ADDR`

The value `ADQ_MARKER_MODE_USER_ADDR` is used together with the data transfer interface (Section 10.4) and implies that markers are transferred to the (user owned) memory at `marker_buffer_bus_address`.

Detecting filled transfer buffers is either done by the user application or via a third-party library by monitoring the marker value for each buffer. Each marker consists of a 32-bit value starting at zero. The first time a buffer is available the value 1 is written to the corresponding marker buffer. Each time new data is available in the buffer the marker value will increase by 1. This marker format is compatible with `clEnqueueWaitSignalAMD()`.

`write_lock_enabled (int32_t)`

Activates a feature that prevents record and metadata buffers from being overwritten before they are returned by user application. Must be set to 1 (default) when the data readout interface (Section 10.5) is used.

If the data transfer interface (Section 10.4) is used. The parameter can be set to 0 to remove the need to call `UnlockP2pBuffers()`, as described in Section 6. Generally, this is *not* recommended unless there are reasons that `UnlockP2pBuffers()` cannot be used and real time processing of buffers is guaranteed.

`transfer_records_to_host_enabled (int32_t)`

When this parameter is set to 1 (default), transfer buffers will be allocated in the host computer's RAM by the API. The parameter *must* be set to 1 when data readout interface (Section 10.5) is used.

If the parameter is set to 0, the user application is responsible for transfer buffer allocation. Manual transfer buffer allocation can only be used with the data transfer interface (Section 10.4). The addresses of the allocated transfer buffers are entered in `record_buffer_bus_address` and `metadata_buffer_bus_address`, if metadata is enabled. Each transfer buffer must be contiguous and available for direct memory access (DMA). User supplied transfer buffers can reside in any or multiple endpoints, including the host computer's RAM.

`packed_buffers_enabled (int32_t)`

If this parameter is set to 0 (default), transfer buffers will be allocated as independent memory regions for all active channels.

If the parameter is set to 1, the API will allocate `nof_buffers` contiguous memory ranges, each corresponding to a transfer buffer index. Each contiguous memory range will contain one transfer buffer for each active channel, placed back-to-back. If metadata is enabled, metadata buffers will

be allocated in the same manner. This allocation scheme is useful in multichannel applications with high throughput and small buffer sizes where the received data is copied to another location like a disk or a GPU. For a given transfer buffer index, data from all active channels can be copied with a single operation, reducing overhead. The source pointers to the packed buffers are found in `record_buffer` and `metadata_buffer` of the lowest active channel index. The size of the packed buffers are found in `record_buffer_packed_size` and `metadata_buffer_packed_size`. The position of the record data for each channel within a packed buffer is found in `record_buffer_packed_offset` and `metadata_buffer_packed_offset`.

When packed buffers are used, `nof_buffers` must be equal for all active channels. The data acquisition and data transfer must be configured so that the transfer buffers for all active channels are filled at the same rate. For triggered acquisition, this typically means that all active channels must use the same `trigger_source` and the `record_buffer_size` must be set to the same multiple of `record_size`.

`dsu_doorbell_value_mask (uint32_t)`

Reserved

`dsu_operation_size (int32_t)`

Reserved

```
struct ADQDataTransferParametersChannel {
    uint64_t            record_buffer_bus_address[ADQ_MAX_NOF_BUFFERS];
    uint64_t            metadata_buffer_bus_address[ADQ_MAX_NOF_BUFFERS];
    uint64_t            marker_buffer_bus_address[ADQ_MAX_NOF_BUFFERS];
    int64_t             nof_buffers;
    int64_t             record_size;
    int64_t             record_buffer_size;
    int64_t             metadata_buffer_size;
    int64_t             record_buffer_packed_offset;
    int64_t             metadata_buffer_packed_offset;
    double              eject_buffer_timeout;
    volatile void     * record_buffer[ADQ_MAX_NOF_BUFFERS];
    volatile void     * metadata_buffer[ADQ_MAX_NOF_BUFFERS];
    volatile uint32_t * marker_buffer[ADQ_MAX_NOF_BUFFERS];
    int32_t             infinite_record_length_enabled;
    int32_t             dynamic_record_length_enabled;
    int32_t             continue_on_overflow_enabled;
    int32_t             record_enabled;
    int32_t             metadata_enabled;
    int32_t             dsu_record_enabled;
    int32_t             dsu_metadata_enabled;
    uint32_t            dsu_record_enabled_endpoints_mask;
    uint32_t            dsu_metadata_enabled_endpoints_mask;
    enum ADQFunction    eject_buffer_source;
}
```

Description

This struct is a member of `ADQDataTransferParameters` and defines the data transfer parameters for a channel.

Members

`record_buffer_bus_address[ADQ_MAX_NOF_BUFFERS]` (`uint64_t`)

Bus addresses to the record transfer buffers. This array is filled in

- by the API if `transfer_records_to_host_enabled` is set to 1; and
- by the user application if `transfer_records_to_host_enabled` is set to 0. Each transfer buffer must be contiguous and available for direct memory access. User supplied transfer buffers can reside in any or multiple endpoints, including the host computer's RAM.

`metadata_buffer_bus_address[ADQ_MAX_NOF_BUFFERS]` (`uint64_t`)

Like `record_buffer_bus_address` but for the metadata transfer buffers.

`marker_buffer_bus_address[ADQ_MAX_NOF_BUFFERS]` (`uint64_t`)

Bus addresses to marker buffers (Section 10.2). This array is filled in

- by the API if `marker_mode` is set to `ADQ_MARKER_MODE_HOST_AUTO` or `ADQ_MARKER_MODE_HOST_MANUAL`; and
- by the user application if `marker_mode` is set to `ADQ_MARKER_MODE_USER_ADDR`. Each marker buffer must be available for direct memory access. User supplied marker buffers can reside in any or multiple endpoints, including the host computer's RAM.

`nof_buffers (int64_t)`

The number of transfer buffers pairs to use, see Section 10.1. For an active channel, valid values are in the range [2, `ADQ_MAX_NOF_BUFFERS`]. Set the parameter to 0 (default) to disable the channel.

`record_size (int64_t)`

The record size in bytes. This parameter is used when the digitizer is configured to acquire records with static length and should be calculated by the user application as

$$\texttt{record\_length} \cdot \texttt{bytes\_per\_sample}.$$

However, when the digitizer is configured

- for records with infinite length,
- for records with dynamic length (Section 9.1); or
- to continue on overflow (Section 10.6.3),

the value must be set to 0 (default).

`record_buffer_size (int64_t)`

The effective record transfer buffer size in bytes. This value must either

1. be a multiple of `record_size`, if larger than or equal to `record_size`; or
2. a multiple of `record_buffer_size_step` if smaller than `record_size`; or
3. a multiple of `record_buffer_size_step` if `record_size` is zero.

The multiple in case 1 must be the same as for `metadata_buffer_size`. Since the buffer allocation always is page aligned, the allocated buffer size may be larger than the effective size. The default value is 0.

`metadata_buffer_size (int64_t)`

The effective record metadata buffer size in bytes. This value must be an multiple of the size of an `ADQGen4RecordHeader`. The multiple must be the same as for `record_buffer_size`. Since the buffer allocation always is page aligned, the allocated buffer size may be larger than the effective size. The default value is 0.

`record_buffer_packed_offset (int64_t)`

The offset of the channel's record data in a packed buffer. Only used when `packed_buffers_enabled` is set to 1. See `packed_buffers_enabled` for additional details.

`metadata_buffer_packed_offset` (`int64_t`)

The offset of the channel's metadata in a packed buffer. Only used when `packed_buffers_enabled` is set to 1. See `packed_buffers_enabled` for additional details.

`eject_buffer_timeout` (`double`)

This parameter specifies a timeout, in seconds, after which any active (partially filled) transfer buffer is ejected. The timeout is enabled when `eject_buffer_timeout` is set to a value greater than zero and disabled by specifying the value zero. The resolution is in the hundreds of microseconds range for the default sampling rate. Moreover, the resolution and the lower and upper bounds of the parameter depends on the digitizer model, its channel configuration and its current `sampling_frequency` but is generally in the range

$$[0, 65535] \cdot \frac{65536 \cdot \{8, 16\}}{\texttt{sampling\_frequency}}.$$

The `eject_buffer_source` must be set to `ADQ_FUNCTION_INVALID` when the timeout is enabled. See Section 10.7 for additional details.

Note

The timeout mechanism is not intended to provide a high-precision timer. For real-time requirements, use one of the pattern generators to define an eject signal. Refer to Section 10.7 for more information.

`record_buffer[ADQ_MAX_NOF_BUFFERS]` (`volatile void *`)

Pointers to the record transfer buffers. Only valid when `transfer_records_to_host_enabled` is set to 1. The default value is `NULL`.

`metadata_buffer[ADQ_MAX_NOF_BUFFERS]` (`volatile void *`)

Pointers to the metadata transfer buffers. Only valid when `transfer_records_to_host_enabled` is set to 1. The default value is `NULL`.

`marker_buffer[ADQ_MAX_NOF_BUFFERS]` (`volatile uint32_t *`)

Pointers to the marker buffers. Only valid when `marker_mode` is set to `ADQ_MARKER_MODE_HOST_AUTO` or `ADQ_MARKER_MODE_HOST_MANUAL`. The default value is `NULL`.

`infinite_record_length_enabled` (`int32_t`)

This parameters specifies whether or not the channel supports the transfer of a record with infinite length, i.e. where `record_length` is set to `ADQ_INFINITE_RECORD_LENGTH`. Refer to Section 10.5.7 for more information on how a record with this property needs to be received by the user application.

`dynamic_record_length_enabled` (`int32_t`)

This parameter specifies whether or not the channel supports the *transfer* of a record with dynamic length, i.e. when the record length is not known beforehand. See Sections 9.1 and 10 for more information.

When enabled, the following must be fulfilled:

- metadata must be enabled by setting `metadata_enabled` to 1,
- the `record_buffer_size` parameter must be set to a multiple of the `record_buffer_size_step` value, and
- the `record_size` parameter must be set to 0.

`continue_on_overflow_enabled (int32_t)`

This parameter specifies whether or not the channel should continue the data acquisition when the on-board memory is full. If disabled (the default value), the data acquisition process will immediately stop once this condition is met. The records already present in the on-board memory may be transferred to the receiving endpoint, but no new records will be acquired. If continue on overflow is enabled, new records will be safely discarded until there's enough space in the on-board memory to fit a *complete* record. When continue on overflow is enabled, the *data transfer* parameter `dynamic_record_length_enabled` must be set to 1.

The default value is 0 (disabled). Refer to Section 10.6 for more information on the different overflow behaviors.

Important

The parameter `continue_on_overflow_enabled` is not applicable for FWATD. FWATD will always safely discard data on overflow, as described in Section 5.6.6.

`record_enabled (int32_t)`

Reserved

`metadata_enabled (int32_t)`

This parameter specifies whether or not record metadata is transferred. The metadata constitutes the basis for the record header `ADQGen4RecordHeader`. Set the parameter to 1 (default) to enable and 0 to disable.

Important

If disabled, the `header` member in a record received from `WaitForRecordBuffer()` will be set to `NULL`. Make sure to not access this member if metadata is disabled.

`dsu_record_enabled (int32_t)`

Reserved

`dsu_metadata_enabled (int32_t)`

Reserved

`dsu_record_enabled_endpoints_mask (uint32_t)`

Reserved

`dsu_metadata_enabled_endpoints_mask (uint32_t)`

Reserved

`eject_buffer_source (enum ADQFunction)`

This parameter specifies the source that will be used to eject a partially filled transfer buffer, see Section 10.7. Valid sources are:

- `ADQ_FUNCTION_INVALID` (default)
- `ADQ_FUNCTION_PATTERN_GENERATOR0`
- `ADQ_FUNCTION_PATTERN_GENERATOR1`
- `ADQ_FUNCTION_RECORD_STOP`

The `eject_buffer_timeout` must be set to zero when an eject buffer source is specified.

```
struct ADQDataReadoutParameters {
    enum ADQParameterId                     id;
    int32_t                                 reserved;
    struct ADQDataReadoutParametersCommon   common;
    struct ADQDataReadoutParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                magic;
}
```

Description

This struct defines the parameters for the data readout process.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DATA_READOUT`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`common (struct ADQDataReadoutParametersCommon)`

A `ADQDataReadoutParametersCommon` struct holding data transfer parameters that apply to all channels.

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQDataReadoutParametersChannel)`

An array of `ADQDataReadoutParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_transfer_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQDataReadoutParametersCommon {
    enum ADQMemoryOwner  memory_owner;
    int32_t              thread_sleep_duration;
}
```

Description

This struct is a member of `ADQDataReadoutParameters` and defines data transfer parameters that apply to all channels. Currently, no ADQ3 series digitizer uses any of the member parameters.

Members

`memory_owner` (`enum ADQMemoryOwner`)

This parameter specifies who is responsible for memory management: the API or the user. Currently, the only supported value is `ADQ_MEMORY_OWNER_API` (default).

`thread_sleep_duration` (`int32_t`)

This parameter specifies the sleep duration, in microseconds, for the internal data readout thread. The thread will sleep this amount between each iteration. The value can be used to limit the CPU load, at the cost of reduced throughput. The default value is 0.

Important

Setting the `thread_sleep_duration` to a positive value will limit the maximum throughput.

```
struct ADQDataReadoutParametersChannel {
    int64_t  nof_record_buffers_max;
    int64_t  nof_record_buffers_in_array;
    int64_t  record_buffer_size_max;
    int64_t  record_buffer_size_increment;
    int32_t  incomplete_records_enabled;
    int32_t  reserved;
}
```

Description

This struct is a member of `ADQDataReadoutParameters` and defines the data readout parameters for a channel.

Members

`nof_record_buffers_max` (`int64_t`)

This parameter specifies an upper limit for the number of record buffers that are *dynamically allocated* for the channel. Any positive number ≥ 0 is allowed, along with the special value `ADQ_INFINITE_NOF_RECORDS` which indicates that there is no upper limit. If set to zero, the dynamic allocation mechanism is disabled. The default value is

- `ADQ_INFINITE_NOF_RECORDS` if the current firmware is FWATD; and

- zero if the current firmware is FWDAQ.

Note

This parameter is currently only used when the digitizer is running the FWATD firmware (Section 5.6).

`nof_record_buffers_in_array (int64_t)`

When this parameter is set to a positive integer, the function `WaitForRecordBuffer()` will emit objects of the type `ADQGen4RecordArray`, grouping together consecutive record buffers and presenting them as a single unit.

When this parameter is set to zero (the default value), `WaitForRecordBuffer()` will instead emit single `ADQGen4Record` objects, i.e. *one record buffer per call*.

Negative values are not allowed, except for the value `ADQ_FOLLOW_RECORD_TRANSFER_BUFFER` (-1) which indicates that the number of array elements should follow the properties of a record transfer buffer.

Important

Changing this parameter from its default value is only required if the data readout interface starts to limit the data thoughput to the user application. See Section 10.5.8 for more information on when this may be required.

`record_buffer_size_max (int64_t)`

This parameter specifies an upper limit for how large a *dynamically allocated* record buffer is allowed to grow (in bytes). Any positive number > 0 is allowed, along with the special number `ADQ_INFINITE_RECORD_LENGTH` which indicates that there is no upper limit. The default value is `ADQ_INFINITE_RECORD_LENGTH`.

Note

This parameter is currently only used when the digitizer is running the FWATD firmware (Section 5.6).

`record_buffer_size_increment (int64_t)`

This parameter specifies the minimum amount (in bytes) by which a record buffer grows when reallocation is required. If allowed by `record_buffer_size_max`, the reallocation will first attempt to grow the buffer by as much as needed, but always at least by the size indicated by `record_buffer_size_increment`. Additionally, the value of this parameter specifies the initial size of a record buffer. The default value is 128 kiB.

Note

This parameter is currently only used when the digitizer is running the FWATD firmware (Section 5.6).

`incomplete_records_enabled (int32_t)`

As described in Section 10.5.7, this parameters determines whether or not incomplete records

are allowed to propagate to the user via `WaitForRecordBuffer()`. The default value is 0.

`reserved (int32_t)`

Reserved

```
struct ADQDbsParameters {
    enum ADQParameterId             id;
    int32_t                         reserved;
    struct ADQDbsParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                        magic;
}
```

Description

This struct defines the parameters of the digital baseline stabilization module for all channels of the digitizer. See Section 5.3 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DBS`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQDbsParametersChannel)`

An array of `ADQDbsParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQDbsParametersChannel {
    int64_t  level;
    int64_t  lower_saturation_level;
    int64_t  upper_saturation_level;
    int32_t  enabled;
    int32_t  reserved;
}
```

Description

This struct is a member of `ADQDbsParameters` and defines digital baseline stabilization parameters for a

channel.

Members

`level` (`int64_t`)
: The target DC level in ADC codes.

`lower_saturation_level` (`int64_t`)
: An advanced paramameter that selects how many codes below the baseline the signal may be before it is ignored in the DC estimation. The value is given as a negative number. Set this parameter to zero to use the default level.

`upper_saturation_level` (`int64_t`)
: An advanced paramameter that selects how many codes above the baseline the signal may be before it is ignored in the DC estimation. The value is given as a positive number. Set this parameter to zero to use the default level.

`enabled` (`int32_t`)
: Set to a nonzero value to enable. The default value is 0 (disabled).

`reserved` (`int32_t`)
: Reserved

```
struct ADQDigitalGainAndOffsetParameters {
    enum ADQParameterId                                id;
    int32_t                                            reserved;
    struct ADQDigitalGainAndOffsetParametersChannel    channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                           magic;
}
```

Description

This struct defines the parameters of the digital gain and offset module for all channels of the digitizer. See Section 5.1 for a high-level description.

Members

`id` (`enum ADQParameterId`)
: The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_DIGITAL_GAINANDOFFSET`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved` (`int32_t`)
: Reserved

`channel[ADQ_MAX_NOF_CHANNELS]` (`struct ADQDigitalGainAndOffsetParametersChannel`)
: An array of `ADQDigitalGainAndOffsetParametersChannel` structs where each element repre-

sents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQDigitalGainAndOffsetParametersChannel {
    int64_t  gain;
    int64_t  offset;
}
```

Description

This struct is a member of `ADQDigitalGainAndOffsetParameters` and defines digital baseline stabilization parameters for a channel.

Members

`gain` (`int64_t`)

The channel's digital gain. The value is normalized to 10 bits, i.e. a value of 1024 corresponds to unity gain. This number is also defined as `ADQ_UNITY_GAIN`. The allowed range is $[-8192, 8192]$ and the default value is `ADQ_UNITY_GAIN`.

`offset` (`int64_t`)

The offset value in ADC codes. The offset is affected by the `gain`, e.g. an offset of 32 codes will shift the data stream by 32 codes multiplied by the normalized gain. The default value is zero.

```
struct ADQEventSourceParameters {
    enum ADQParameterId                          id;
    int32_t                                      reserved;
    struct ADQEventSourcePeriodicParameters      periodic;
    struct ADQEventSourceLevelParameters         level;
    struct ADQEventSourceLevelMatrixParameters   level_matrix;
    struct ADQEventSourcePortParameters          port[ADQ_MAX_NOF_PORTS];
    struct ADQEventSourceMatrixParameters        matrix;
    uint64_t                                     magic;
}
```

Description

This is a high-level struct collecting the parameters of *all* the event sources of the digitizer. This means that each member struct is *also* an object that may interact with the configuration functions (see Section A.4.3). Refer to Section 6 for a high-level description of event sources.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`periodic (struct ADQEventSourcePeriodicParameters)`

A `ADQEventSourcePeriodicParameters` struct holding the parameters of the periodic event source. See Section 6.3 for a high-level description.

`level (struct ADQEventSourceLevelParameters)`

A `ADQEventSourceLevelParameters` struct holding the parameters of the signal level event sources. See Section 6.4 for a high-level description.

`level_matrix (struct ADQEventSourceLevelMatrixParameters)`

A `ADQEventSourceLevelMatrixParameters` struct holding the parameters of the signal level event source matrix. See Section 6.5 for a high-level description.

`port[ADQ_MAX_NOF_PORTS] (struct ADQEventSourcePortParameters)`

An array of `ADQEventSourcePortParameters` structs where each element represents the parameters for ports with an associated event source. Not all ports have an event source tied to them. This is indicated by the constant parameter `event_source`. The array is intended to be indexed by using the enumeration `ADQPort`.

`matrix (struct ADQEventSourceMatrixParameters)`

An `ADQEventSourceMatrixParameters` struct holding the parameters of the matrix event source. See Section 6.10 for a high-level description.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourceLevelParameters {
    enum ADQParameterId                          id;
    int32_t                                      reserved;
    struct ADQEventSourceLevelParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                     magic;
}
```

Description

This struct defines the parameters of the signal level event sources for all channels of the digitizer. See

Section 6.4 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQEventSourceLevelParametersChannel)`

An array of `ADQEventSourceLevelParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_acquisition_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourceLevelParametersChannel {
    int64_t  level;
    int64_t  arm_hysteresis;
}
```

Description

This struct is a member of `ADQEventSourceLevelParameters` and defines the signal level event source parameters for a channel.

Members

`level (int64_t)`

The signal threshold in ADC codes. The default value is 0.

`arm_hysteresis (int64_t)`

The arm hysteresis in ADC codes. The default value is 100.

```
struct ADQEventSourceLevelMatrixParameters {
    enum ADQParameterId                                 id;
    int32_t                                             reserved;
    struct ADQEventSourceLevelMatrixParametersChannel   channel;
    uint64_t                                            magic;
}
```

Description

This struct is a member of `ADQEventSourceLevelParameters` and defines the parameters for the signal level matrix event source.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL_MATRIX`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel (struct ADQEventSourceLevelMatrixParametersChannel)`

An array of `ADQEventSourceLevelMatrixParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_acquisition_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourceLevelMatrixParametersChannel {
    int32_t       enabled;
    enum ADQEdge  edge;
}
```

Description

This struct is a member of `ADQEventSourceLevelMatrixParameters` and holds the parameters for the channel's signal level event source in the matrix.

Members

`enabled (int32_t)`

This parameter specifies whether or not to enable the event stream from the target channel in the matrix. Set to a nonzero value to enable, zero to disable. The default value is 0.

`edge` (`enum ADQEdge`)

The edge sensitivity for the target channel event source. Valid values are:

- `ADQ_EDGE_RISING`
- `ADQ_EDGE_FALLING`
- `ADQ_EDGE_BOTH`

When the channel is not enabled the edge sensitivity is ignored. The default value is `ADQ_EDGE_RISING`.

```
struct ADQEventSourcePeriodicParameters {
    enum ADQParameterId  id;
    enum ADQEventSource  synchronization_source;
    int64_t              period;
    int64_t              high;
    int64_t              low;
    double               frequency;
    uint64_t             magic;
}
```

Description

This struct defines the parameters of the periodic event source. See Section 6.3 for a high-level description. The event source offers three different methods of configuring the properties of the underlying digital periodic signal, specifying either (in order of precedence):

- the logic `high` and logic `low` durations,
- the `period`; or
- the `frequency`.

The first nonzero value will determine which method is used to specify the properties of the signal.

Important

Reading the current parameters via `GetParameters()` will set *all* the parameters to nonzero values corresponding to the current configuration. For example, setting the frequency to 1 kHz and reading back the result will result in `high`, `low` and `period` being set to the values corresponding to a periodic signal with frequency 1 kHz.

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE_PERIODIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`synchronization_source` (`enum ADQEventSource`)

Reserved

`period` (`int64_t`)

The period of the signal, given as a whole number of sampling periods (measured in the digitizer's `sampling_frequency`). This value takes precedence over `frequency` but is only used if both `high` and `low` are set to zero. The default value is zero.

`high` (`int64_t`)

The duration of the logic high part of the periodic signal, given as a whole number of sampling periods (measured in the digitizer's `sampling_frequency`). This value takes precedence over both `frequency` and `period`. The default value is zero.

`low` (`int64_t`)

The duration of the logic low part of the periodic signal, given as a whole number of sampling periods (measured in the digitizer's `sampling_frequency`). This value takes precedence over both `frequency` and `period`. The default value is zero.

`frequency` (`double`)

The frequency of the periodic signal, in Hertz. This value is only used if the other parameters are set to zero. The precision of the frequency approach depends on the `sampling_frequency` of the digitizer. The closest synthesizable frequency will be selected. The resulting frequency is readable by calling `GetParameters()`. The default value is 1000.0.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourceSoftwareParameters {
    enum ADQParameterId  id;
    int32_t              reserved;
    uint64_t             magic;
}
```

Description

This struct defines the parameters of the software controlled event source.

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE_SOFTWARE`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourcePortParameters {
    enum ADQParameterId                     id;
    int32_t                                 reserved;
    struct ADQEventSourcePortParametersPin  pin[ADQ_MAX_NOF_PINS];
    uint64_t                                magic;
}
```

Description

This struct defines the parameters of the event source associated with a port. See Sections 6.6–6.7 for a high-level description. While the parameter definition is *shared* across all ports, the event sources associated with them are *distinct*. A consequence of this is that the struct identifier, `id`, may have several valid values, each targeting the event source of a specific port.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to one of the following values:

- `ADQ_PARAMETER_ID_EVENT_SOURCE_TRIG`
- `ADQ_PARAMETER_ID_EVENT_SOURCE_SYNC`
- `ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOA`
- `ADQ_PARAMETER_ID_EVENT_SOURCE_GPIOB`
- `ADQ_PARAMETER_ID_EVENT_SOURCE_PXIE`

This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`pin[ADQ_MAX_NOF_PINS] (struct ADQEventSourcePortParametersPin)`

An array of `ADQEventSourcePortParametersPin` structs where each element represents the parameters of a pin in the port. Some ports only have one pin (index 0) and some ports may have several. Refer to the constant parameter `nof_pins` for the corresponding port to programmatically determine the number of available pins.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the

parameter set.

```
struct ADQEventSourcePortParametersPin {
    int32_t       reference_clock_synchronization_enabled;
    double        threshold;
    enum ADQEdge  reference_clock_synchronization_edge;
}
```

Description

This struct is a member of `ADQEventSourcePortParameters` and defines the parameters of a pin. See Sections 6.6–6.7 for a high-level description.

Members

`reference_clock_synchronization_enabled (int32_t)`

If set to 1 the event source will be synchronized to the reference clock.

`threshold (double)`

The threshold in Volts. The default value is 0.5 V for `ADQ_PARAMETER_ID_EVENT_SOURCE_TRIG` and `ADQ_PARAMETER_ID_EVENT_SOURCE_SYNC`. For other event sources, this parameter is unused.

`reference_clock_synchronization_edge (enum ADQEdge)`

The event source edge that should be synchronized. If a single edge is selected the other edge will be discarded. The default value is `ADQ_EDGE_BOTH`.

```
struct ADQEventSourceMatrixParameters {
    enum ADQParameterId                          id;
    int32_t                                      reserved;
    struct ADQEventSourceMatrixParametersInput   input[ADQ_MAX_NOF_MATRIX_INPUTS];
    uint64_t                                     magic;
}
```

Description

This struct defines the parameters of the matrix event source. See Section 6.10 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_EVENT_SOURCE_MATRIX`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`input[ADQ_MAX_NOF_MATRIX_INPUTS]` (`struct ADQEventSourceMatrixParametersInput`)

An array of `ADQEventSourceMatrixParametersInput` structs where each element defines one event source and corresponding edges (rising, falling, both) that should be considered by this source.

If multiple of the configured event sources triggers an event during the same data clock cycle, the one with lowest index will have priority.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQEventSourceMatrixParametersInput {
    enum ADQEventSource  source;
    enum ADQEdge         edge;
}
```

Description

This struct defines the parameters of each event source connected to the matrix event source and is part of `ADQEventSourceMatrixParameters`.

Members

`source` (`enum ADQEventSource`)

An event source. Valid options are:

- `ADQ_EVENT_SOURCE_TRIG`
- `ADQ_EVENT_SOURCE_SOFTWARE`
- `ADQ_EVENT_SOURCE_GPIOA0`
- `ADQ_EVENT_SOURCE_GPIOB0`
- `ADQ_EVENT_SOURCE_SYNC`
- `ADQ_EVENT_SOURCE_PXIE_STARB`

The special case `ADQ_EVENT_SOURCE_INVALID` is also valid and means no source/disabled. The default value is `ADQ_EVENT_SOURCE_INVALID`.

`edge` (`enum ADQEdge`)

An event edge. Valid options are:

- `ADQ_EDGE_RISING`
- `ADQ_EDGE_FALLING`
- `ADQ_EDGE_BOTH`

The default value is `ADQ_EDGE_RISING`.

```
struct ADQFirFilterParameters {
    enum ADQParameterId                  id;
    int32_t                              reserved;
    struct ADQFirFilterParametersChannel channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                             magic;
}
```

Description

This struct defines the parameters of the FIR filter module for all channels of the digitizer. See Section 5.4 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_FIR_FILTER`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQFirFilterParametersChannel)`

An array of `ADQFirFilterParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQFirFilterParametersChannel {
    enum ADQRoundingMethod     rounding_method;
    enum ADQCoefficientFormat  format;
    double                     coefficient[ADQ_MAX_NOF_FILTER_COEFFICIENTS];
    int32_t                    coefficient_fixed_point[ADQ_MAX_NOF_FILTER_COEFFICIENTS];
}
```

Description

This struct is a member of `ADQFirFilterParameters` and defines the FIR filter parameters for a channel.

Members

`rounding_method (enum ADQRoundingMethod)` Write-only

The rounding method that is used to convert the floating point values in the `coefficient` array to the fixed point precision of the filter. The default value is `ADQ_ROUNDING_METHOD_TIE_AWAY_FROM_ZERO`. This parameter is write-only.

`format` (`enum ADQCoefficientFormat`) — Write-only

The coefficient format to use when setting the filter coefficients. Refer to the enumeration `ADQCoefficientFormat` for more information. The default value is `ADQ_COEFFICIENT_FORMAT_DOUBLE`. This parameter is write-only.

`coefficient[ADQ_MAX_NOF_FILTER_COEFFICIENTS]` (`double`)

The filter coefficients, in double-precision floating point format. When setting the parameters of the filter, this array is only used if the `format` is set to `ADQ_COEFFICIENT_FORMAT_DOUBLE`.

`coefficient_fixed_point[ADQ_MAX_NOF_FILTER_COEFFICIENTS]` (`int32_t`)

The filter coefficients, in fixed point format. When setting the parameters of the filter, this array is only used if the `format` is set to `ADQ_COEFFICIENT_FORMAT_FIXED_POINT`.

```
struct ADQFunctionParameters {
    enum ADQParameterId                          id;
    int32_t                                      reserved;
    struct ADQPatternGeneratorParameters         pattern_generator[ADQ_MAX_NOF_PATTERN_GENERATORS];
    struct ADQPulseGeneratorParameters           pulse_generator[ADQ_MAX_NOF_PULSE_GENERATORS];
    struct ADQTimestampSynchronizationParameters timestamp_synchronization;
    struct ADQDaisyChainParameters               daisy_chain;
    uint64_t                                     magic;
}
```

Description

This is a high-level struct collecting the parameters of *all* the function modules of the digitizer. This means that each member struct is *also* an object that may interact with the configuration functions (see Section A.4.3). Refer to Section 7 for a high-level description of the various function modules.

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_FUNCTION`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved` (`int32_t`)

Reserved

`pattern_generator[ADQ_MAX_NOF_PATTERN_GENERATORS]` (`struct ADQPatternGeneratorParameters`)

An array of `ADQPatternGeneratorParameters` structs where each element represents the parameters for one pattern generator (see Section 7.1). The number of valid/active entries is given by the constant parameter `nof_pattern_generators`.

`pulse_generator[ADQ_MAX_NOF_PULSE_GENERATORS]` (`struct ADQPulseGeneratorParameters`)

An array of `ADQPulseGeneratorParameters` structs where each element represents the parameters for one pulse generator (see Section 7.2). The number of valid/active entries is given by the constant parameter `nof_pulse_generators`.

`timestamp_synchronization` (`struct ADQTimestampSynchronizationParameters`)

A `ADQTimestampSynchronizationParameters` struct holding the parameters of the timestamp synchronization function. See Section 7.3 for a high-level description.

`daisy_chain` (`struct ADQDaisyChainParameters`)

A `ADQDaisyChainParameters` struct holding the parameters of the daisy chain function. See Section 7.4 for a high-level description.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQPatternGeneratorParameters {
    enum ADQParameterId                     id;
    int32_t                                 nof_instructions;
    struct ADQPatternGeneratorInstruction   instruction[ADQ_MAX_NOF_PATTERN_INSTRUCTIONS];
    uint64_t                                magic;
}
```

Description

This struct defines the parameters for the pattern generator. See Section 7.1 for a high-level description. There may be more than one pattern generator available, determined by `nof_pattern_generators`. While the parameter definition is *shared* across all generators, they each have a distinct struct identifier (`id`).

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to one of the following values:

- `ADQ_PARAMETER_ID_PATTERN_GENERATOR0`
- `ADQ_PARAMETER_ID_PATTERN_GENERATOR1`

This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`nof_instructions` (`int32_t`) Write-only

The number of pattern generator instructions. Valid values are $0, 1, ..., 16$. Setting `nof_instructions` to zero will disable the pattern generator. This parameter is write-only.

`instruction[ADQ_MAX_NOF_PATTERN_INSTRUCTIONS]` (`struct ADQPatternGeneratorInstruction`) Write-only

An array of `ADQPatternGeneratorInstruction` structs where each element represents an instruction. This parameter is write-only.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQPatternGeneratorInstruction {
    int64_t                          count;
    int64_t                          count_prescaling;
    enum ADQPatternGeneratorOperation  op;
    enum ADQEventSource              source;
    enum ADQEdge                     source_edge;
    enum ADQEventSource              reset_source;
    enum ADQEdge                     reset_source_edge;
    int32_t                          output_value;
    int32_t                          output_value_transition;
    int32_t                          reserved;
}
```

Description

This struct defines one pattern generator instruction.

Members

`count` (`int64_t`)

The number of events or time before the next instruction is loaded. The behavior of this parameter depends on the `op` parameter:

`ADQ_PATTERN_GENERATOR_OPERATION_TIMER`

The count specifies the *time* in sampling periods until the next instruction is loaded. The valid range depends on the digitizer model and its current firmware:

- `ADQ30`
  - `1CH-FWDAQ`, `1CH-FWATD`: valid range of $[8, 2^{35} - 8]$, step size of 8
- `ADQ32, ADQ33`
  - `2CH-FWDAQ`, `2CH-FWATD`: valid range of $[8, 2^{35} - 8]$, step size of 8
  - `1CH-FWDAQ`, `1CH-FWATD`: valid range of $[16, 2^{36} - 16]$, step size of 16
- `ADQ36`

- `4CH-FWDAQ`: valid range of $[8, 2^{35} - 8]$, step size of 8
- `2CH-FWDAQ`: valid range of $[16, 2^{36} - 16]$, step size of 16

`ADQ_PATTERN_GENERATOR_OPERATION_EVENT`

The count specified the *number of events* of the selected source until the next instruction is loaded. Valid values are $1, ..., 2^{32} - 1$.

`count_prescaling (int64_t)`

Prescaling of the `count` parameter. Valid values are $1, ..., 255$. The default value is 1 (no scaling).

`op (enum ADQPatternGeneratorOperation)`

The instruction operation. Valid values are:

- `ADQ_PATTERN_GENERATOR_OPERATION_EVENT`
- `ADQ_PATTERN_GENERATOR_OPERATION_TIMER`

The default value is `ADQ_PATTERN_GENERATOR_OPERATION_TIMER`.

`source (enum ADQEventSource)`

The instruction event source. Only used if then operation is set to `ADQ_PATTERN_GENERATOR_OPERATION_EVENT`. Valid values are:

- `ADQ_EVENT_SOURCE_INVALID`
- `ADQ_EVENT_SOURCE_SOFTWARE`
- `ADQ_EVENT_SOURCE_TRIG`
- `ADQ_EVENT_SOURCE_PERIODIC`
- `ADQ_EVENT_SOURCE_SYNC`
- `ADQ_EVENT_SOURCE_GPIOA0`
- `ADQ_EVENT_SOURCE_GPIOB0`
- `ADQ_EVENT_SOURCE_PXIE_STARB`
- `ADQ_EVENT_SOURCE_PXIE_TRIG0`
- `ADQ_EVENT_SOURCE_PXIE_TRIG1`
- `ADQ_EVENT_SOURCE_REFERENCE_CLOCK`
- `ADQ_EVENT_SOURCE_MATRIX`

The default value is `ADQ_EVENT_SOURCE_INVALID`.

`source_edge (enum ADQEdge)`

An `ADQEdge` which specifies the edge selection of the `source`. The default value is `ADQ_EDGE_RISING`. Only used if the operation is set to `ADQ_PATTERN_GENERATOR_OPERATION_EVENT`.

`reset_source (enum ADQEventSource)`

The reset source of the instruction, see Section 7.1. Valid values are:

- `ADQ_EVENT_SOURCE_INVALID`
- `ADQ_EVENT_SOURCE_SOFTWARE`

- ADQ_EVENT_SOURCE_TRIG
- ADQ_EVENT_SOURCE_PERIODIC
- ADQ_EVENT_SOURCE_SYNC
- ADQ_EVENT_SOURCE_GPIOA0
- ADQ_EVENT_SOURCE_GPIOB0
- ADQ_EVENT_SOURCE_PXIE_STARB
- ADQ_EVENT_SOURCE_REFERENCE_CLOCK
- ADQ_EVENT_SOURCE_MATRIX

Only used if the operation is set to ADQ_PATTERN_GENERATOR_OPERATION_EVENT. The default value is ADQ_EVENT_SOURCE_INVALID which disables the reset.

`reset_source_edge (enum ADQEdge)`

An ADQEdge which specifies the edge selection of the reset_source. The default value is ADQ_EDGE_RISING. Only used if the operation is set to ADQ_PATTERN_GENERATOR_OPERATION_EVENT.

`output_value (int32_t)`

The value which the pattern generator will output during the instruction. Valid values are 0, 1. The default value is 0.

`output_value_transition (int32_t)`

The value which the pattern generator will output during the last cycle of the instruction. Valid values are 0, 1. The default value is 0.

`reserved (int32_t)`

Reserved

```
struct ADQPdParameters {
    enum ADQParameterId            id;
    int32_t                        reserved;
    struct ADQPdParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                       magic;
}
```

Description

This struct defines the parameters of the PD signal processing module for all channels of the digitizer. See Section 5.7 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to ADQ_PARAMETER_ID_PD. This is guaranteed if InitializeParameters() is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS]` (`struct ADQPdParametersChannel`)

An array of `ADQPdParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_transfer_channels` holds the number of valid entries.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQPdParametersChannel {
    enum ADQPolarity  polarity;
    int32_t           area_leading_edge_window_length;
    int32_t           area_trailing_edge_window_length;
    int32_t           baseline;
}
```

Description

This struct is a member of `ADQPdParameters` and defines the PD signal processing module parameters for a channel. See Section 5.7 for more information.

Members

`polarity` (`enum ADQPolarity`)

The value of this parameter should match the polarity of the unipolar pulses in the system. Valid values are

- `ADQ_POLARITY_POSITIVE` (the default value); and
- `ADQ_POLARITY_NEGATIVE`.

`area_leading_edge_window_length` (`int32_t`)

The length of the leading edge window used when calculating the `area` of a pulse. The area calculation will include the contributions from this number of samples *before* the leading edge crossing event. The valid range is $[0, 64]$ and the default value is 0. See Section 5.7.7 for more information.

`area_trailing_edge_window_length` (`int32_t`)

The length of the trailing edge window used when calculating the `area` of a pulse. The area calculation will include the contributions from this number of samples *after* the trailing edge crossing event. The valid range is $[0, 64]$ and the default value is 0. See Section 5.7.7 for more information.

`baseline` (`int32_t`)

The reference level to use when calculating the attributes of a pulse. The baseline is specified in

ADC codes in the range defined by the channel's `code_normalization` value as

$$\left[-\frac{\texttt{code\_normalization}}{2}, \frac{\texttt{code\_normalization}}{2} - 1\right]$$

The default value is 0.

Important

A correctly configured baseline is critically important for pulse analysis. If the digital baseline stabilization is enabled (Section 5.3), the baseline should be set to the same target `level`.

Refer to Sections 5.7.5–5.7.7 for additional details.

```
struct ADQPdrxParameters {
    enum ADQParameterId              id;
    int32_t                          reserved;
    struct ADQPdrxParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                         magic;
}
```

Description

This struct defines the parameters of the pulse detection range extension module for all channels of the digitizer. See Section 5.5 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_PDRX`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQPdrxParametersChannel)`

An array of `ADQPdrxParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQPdrxParametersChannel {
    int32_t  enabled;
    int32_t  polarity;
    int32_t  equalizer_enabled;
    int32_t  reflection_filter_enabled;
    int32_t  ac_compensation_enabled;
    int32_t  reflection_delay;
    int64_t  dc_offset;
    double   gain;
    double   ac_compensation_filter_zero;
    double   ac_compensation_filter_pole;
    double   ac_compensation_pulse_level;
    double   ac_compensation_pulse_steepness;
    double   equalizer[ADQ_MAX_NOF_PDRX_EQUALIZER_COEFFICIENTS];
    double   reflection_filter[ADQ_MAX_NOF_PDRX_REFLECTION_FILTER_COEFFICIENTS];
}
```

Description

This struct is a member of `ADQPdrxParameters` and defines pulse detection range extension parameters for a channel.

Members

`enabled (int32_t)`

For a channel that supports PDRX, this parameter enables or disables the core PDRX behavior (see Section 5.5). Whether or not a channel supports PDRX is indicated by the channel-specific constant parameter `is_present`. The other parameters in this group are only set (and validated) if this parameter is set to a nonzero value. The default value is zero (disabled).

When activated, the channel's normal data stream is replaced with a new stream that incorporates data from the neighboring `high_gain_channel` to achieve a higher dynamic range. The user should no longer acquire data from the high gain channel when PDRX is active (although it is still technically possible to do so).

The PDRX-specific signal processing blocks each have their own activation parameter in this group. However, all of them require that PDRX is `enabled` for the signal processing to take effect.

`polarity (int32_t)`

This parameter specifies the polarity of the pulses. It should be set to `ADQ_POLARITY_POSITIVE` for positive pulses and to `ADQ_POLARITY_NEGATIVE` for negative pulses. If the assumption about unipolar pulses is not applicable, set the value to `ADQ_POLARITY_INVALID` to combine the channels using symmetric limits. Refer to Section 5.5.2 for details.

`equalizer_enabled (int32_t)`

Set this parameter to a nonzero value to enable the PDRX equalizer (Section 5.5.3). The parameter `enabled` must also be set to a nonzero value. Otherwise, the value is ignored and the equalizer is disabled. The `equalizer` coefficients are only set (and validated) if this parameter is

nonzero. The default value is zero (disabled).

`reflection_filter_enabled (int32_t)`

Set this parameter to a nonzero value to enable the PDRX reflection filter (Section 5.5.4). The parameter `enabled` must also be set to a nonzero value. Otherwise, the value is ignored and the reflection filter is disabled. The `reflection_delay` and the `reflection_filter` coefficients are only set (and validated) if this parameter is nonzero. The default value is zero (disabled).

`ac_compensation_enabled (int32_t)`

Set this parameter to a nonzero value to enable the PDRX AC-coupling (Section 5.5.5) compensation. The parameter `enabled` must also be set to a nonzero value. Otherwise, the value is ignored and the AC-coupling compensation is disabled. The parameters

- `ac_compensation_filter_zero`
- `ac_compensation_filter_pole`
- `ac_compensation_pulse_level`
- `ac_compensation_pulse_steepness`

are only set (and validated) if this parameter is nonzero. The default value is zero (disabled).

`reflection_delay (int32_t)`

This parameter sets the delay (in samples) between the signal and the reflection filter output. The valid range is [12, 255] and the value should be determined by the calibration process described in Section 5.5.6. The parameter is only set (and validated) when `enabled` and `reflection_filter_enabled` are set to nonzero values. The default value is zero.

`dc_offset (int64_t)`

If any analog `dc_offset` is applied, this parameter must be set to the ADC code that corresponds to 0 V. For 16-bit data, this is calculated as

$$\frac{\texttt{dc\_offset}}{\texttt{input\_range}} \cdot 2^{16},$$

using the values from the channel-specific analog front-end parameters.

`gain (double)`

This parameter is used to correctly scale the data when combining the data streams from the high gain and low gain channels. The value should be set to the gain difference between the two channels and is ideally determined by the calibration process described in Section 5.5.6. The parameter is only set (and validated) when `enabled` is set to a nonzero value. The default value is zero. See Section 5.5.2 for more information.

`ac_compensation_filter_zero (double)`

This parameter controls the zero of the AC-coupling compensation's IIR filter (Section 5.5.5). The value should be determined by the calibration process described in Section 5.5.6. The parameter is only set (and validated) when `enabled` and `ac_compensation_enabled` are set to nonzero values. The default value is zero.

`ac_compensation_filter_pole` (`double`)

This parameter controls the pole of the AC-coupling compensation's IIR filter (Section 5.5.5). The value should be determined by the calibration process described in Section 5.5.6. The parameter is only set (and validated) when `enabled` and `ac_compensation_enabled` are set to nonzero values. The default value is zero.

`ac_compensation_pulse_level` (`double`)

This parameter is used (together with `ac_compensation_pulse_steepness`) by the AC-coupling compensation's pulse removal module to condition the signal for baseline estimation. The parameter is only set (and validated) when `enabled` and `ac_compensation_enabled` are set to nonzero values. The default value is zero. See Section 5.5.5 for more information.

`ac_compensation_pulse_steepness` (`double`)

This parameter is used by the AC-coupling compensation's pulse removal module together with `ac_compensation_pulse_level` to condition the signal for baseline estimation. The parameter is only set (and validated) when `enabled` and `ac_compensation_enabled` are set to nonzero values. The default value is zero. See Section 5.5.5 for more information.

`equalizer[ADQ_MAX_NOF_PDRX_EQUALIZER_COEFFICIENTS]` (`double`)

This parameter is an array that specifies the equalizer coefficients in double-precision floating point format. Each coefficient must be in the range $[-2.0, 2.0 - 2^{-14}]$ and is subject to rounding to fit a signed, fixed point Q2.14 format. The number of valid entries is given by the channel-specific constant parameter `nof_equalizer_coefficients`.

The coefficients should be determined by the calibration process described in Section 5.5.6. The array contents is only used (and validated) when `enabled` and `equalizer_enabled` are set to nonzero values. The default coefficient value is zero. See Section 5.5.3 for more information.

`reflection_filter[ADQ_MAX_NOF_PDRX_REFLECTION_FILTER_COEFFICIENTS]` (`double`)

This parameter is an array that specifies the reflection filter coefficients in double-precision floating point format. Each coefficient must be in the range $[-2.0, 2.0 - 2^{-14}]$ and is subject to rounding to fit a signed, fixed point Q2.14 format. The number of valid entries is given by the channel-specific constant parameter `nof_reflection_filter_coefficients`.

The coefficients should be determined by the calibration process described in Section 5.5.6. The array contents is only used (and validated) when `enabled` and `reflection_filter_enabled` are set to nonzero values. The default coefficient value is zero. See Section 5.5.4 for more information.

```
struct ADQPortParameters {
    enum ADQParameterId          id;
    int32_t                      reserved;
    struct ADQPortParametersPin  pin[ADQ_MAX_NOF_PINS];
    uint64_t                     magic;
}
```

Description

This structs defines the parameters of a port. See Section 8 for a high-level description. The parameter definition is *shared* across all ports, but each one has a distinct struct identifier (`id`).

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to one of the following values:

- `ADQ_PARAMETER_ID_PORT_TRIG`
- `ADQ_PARAMETER_ID_PORT_SYNC`
- `ADQ_PARAMETER_ID_PORT_SYNCO`
- `ADQ_PARAMETER_ID_PORT_SYNCI`
- `ADQ_PARAMETER_ID_PORT_CLK`
- `ADQ_PARAMETER_ID_PORT_CLKI`
- `ADQ_PARAMETER_ID_PORT_CLKO`
- `ADQ_PARAMETER_ID_PORT_GPIOA`
- `ADQ_PARAMETER_ID_PORT_GPIOB`
- `ADQ_PARAMETER_ID_PORT_PXIE`
- `ADQ_PARAMETER_ID_PORT_MTCA`

This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

Important

There is no digitizer that features every port in the list above. Refer to the corresponding entry in the constant parameter array `port` to programmatically determine the availability.

`reserved` (`int32_t`)

Reserved

`pin[ADQ_MAX_NOF_PINS]` (`struct ADQPortParametersPin`)

An array of `ADQPortParametersPin` structs where each element represents the parameters of a pin in the port. Some ports only have one pin (index 0) and some ports may have several. Refer to the constant parameter `nof_pins` for the corresponding port to programmatically determine the number of available pins.

`magic` (`uint64_t`)

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the

parameter set.

```
struct ADQPortParametersPin {
    enum ADQImpedance  input_impedance;
    enum ADQDirection  direction;
    enum ADQFunction   function;
    int32_t            value;
    int32_t            invert_output;
    int32_t            reserved;
}
```

Description

This struct is a member of `ADQPortParameters` and defines the parameters of a pin. See Section 8 for a high-level description.

Members

`input_impedance (enum ADQImpedance)`

When the pin is configured as an input, this parameter determines the input impedance. The default value depends on the port and pin.

> **Note**
> Not all pins support a configurable input impedance.

`direction (enum ADQDirection)`

The I/O configuration of the pin. Not all pins support a configurable direction. For those that do, `ADQ_DIRECTION_IN` configures the pin as an input and `ADQ_DIRECTION_OUT` configures the pin as an output. When the output buffer is activated, the digitizer immediately begins driving the digital output signal of whichever `function` is selected. The default value is `ADQ_DIRECTION_IN`.

> **Note**
> Not all pins support a configurable direction.

`function (enum ADQFunction)`

The function selection that determines the output signal when the `direction` is set to `ADQ_DIRECTION_OUT`. Not all functions are able to be selected by all the pins. Refer to Section 8 for a list of which functions each pin supports. The default value is `ADQ_FUNCTION_INVALID`.

`value (int32_t)`

This parameter behaves differently depending on the configuration context:

`GetParameters()`

If the pin has input capabilities, the value will reflect the digital signal level of the pin: 0 for logic low and 1 for logic high. This is true even if the pin is configured as an output, as long as the pin *can* be configured as an input.

`SetParameters()`

If the pin is configured as an output and its `function` is set to `ADQ_FUNCTION_GPIO`, the value of this parameter will set the digital signal level of the pin. Specify 0 for logic low and 1 for logic high. If the prerequisites are not met, the parameter is ignored.

`invert_output (int32_t)`

A boolean value indicating that the digital output signal should be inverted.

`reserved (int32_t)`

Reserved

```
struct ADQPulseGeneratorParameters {
    enum ADQParameterId  id;
    enum ADQEventSource  source;
    enum ADQEdge         edge;
    int32_t              reserved;
    int64_t              length;
    uint64_t             magic;
}
```

Description

This struct defines the parameters for the pulse generator. The pulse generator is disabled when `source` is set to `ADQ_EVENT_SOURCE_INVALID`. See Section 7.2.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to one of the following values:

- `ADQ_PARAMETER_ID_PULSE_GENERATOR0`
- `ADQ_PARAMETER_ID_PULSE_GENERATOR1`
- `ADQ_PARAMETER_ID_PULSE_GENERATOR2`
- `ADQ_PARAMETER_ID_PULSE_GENERATOR3`

This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`source (enum ADQEventSource)`

An `ADQEventSource` whose events are used to trigger a pulse. The default value is `ADQ_EVENT_SOURCE_INVALID`. Not every event source can be used as a source. Valid values are:

- `ADQ_EVENT_SOURCE_SOFTWARE`
- `ADQ_EVENT_SOURCE_TRIG`
- `ADQ_EVENT_SOURCE_LEVEL`
- `ADQ_EVENT_SOURCE_PERIODIC`
- `ADQ_EVENT_SOURCE_SYNC`
- `ADQ_EVENT_SOURCE_GPIOA0`

- ADQ_EVENT_SOURCE_GPIOB0
- ADQ_EVENT_SOURCE_PXIE_STARB
- ADQ_EVENT_SOURCE_PXIE_TRIG0
- ADQ_EVENT_SOURCE_PXIE_TRIG1
- ADQ_EVENT_SOURCE_REFERENCE_CLOCK
- ADQ_EVENT_SOURCE_MATRIX
- ADQ_EVENT_SOURCE_LEVEL_MATRIX

`edge (enum ADQEdge)`

An ADQEdge which specifies the edge selection of the source. The default value is ADQ_EDGE_RISING.

`reserved (int32_t)`

Reserved

`length (int64_t)`

The length of the pulse in samples. Setting the length to -1 will generate a pulse with length equal to that of the source. The valid range depends on the digitzer model and its current firmware:

- ADQ30
  - 1CH-FWDAQ, 1CH-FWATD: valid range of $-1, 8, 16, ..., 2^{19} - 8$
- ADQ32, ADQ33
  - 2CH-FWDAQ, 2CH-FWATD: valid range of $-1, 8, 16, ..., 2^{19} - 8$
  - 1CH-FWDAQ, 1CH-FWATD: valid range of $-1, 16, 32, ..., 2^{20} - 16$
- ADQ36
  - 4CH-FWDAQ: valid range of $-1, 8, 16, ..., 2^{19} - 8$
  - 2CH-FWDAQ: valid range of $-1, 16, 32, ..., 2^{20} - 16$

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to ADQ_PARAMETERS_MAGIC. This is guaranteed if InitializeParameters() is called to initialize the parameter set.

```
struct ADQSampleSkipParameters {
    enum ADQParameterId                   id;
    int32_t                               reserved;
    struct ADQSampleSkipParametersChannel channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                              magic;
}
```

Description

This struct defines the parameters of the sample skip module for all channels of the digitizer. See Section 5.2 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_SAMPLE_SKIP`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQSampleSkipParametersChannel)`

An array of `ADQSampleSkipParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_acquisition_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQSampleSkipParametersChannel {
    int64_t  skip_factor;
}
```

Description

This struct is a member of `ADQSampleSkipParameters` and defines sample skip parameters for a channel.

Members

`skip_factor (int64_t)`

The sample skip factor. The default value is 1, which implies no skipping. The valid range depends on the digitizer model and its current firmware:

- ADQ30
  - 1CH-FWDAQ, 1CH-FWATD: $1, 2, 4, 5, 8, 9, 10, ..., 2^{22} - 1$
- ADQ32, ADQ33
  - 2CH-FWDAQ, 2CH-FWATD: $1, 2, 4, 5, 8, 9, 10, ..., 2^{22} - 1$
  - 1CH-FWDAQ, 1CH-FWATD: $1, 2, 4, 5, 8, 16, 17, 18, ..., 2^{22} - 1$
- ADQ36
  - 4CH-FWDAQ: $1, 2, 4, 5, 8, 9, 10, ..., 2^{22} - 1$
  - 2CH-FWDAQ: $1, 2, 4, 5, 8, 16, 17, 18, ..., 2^{22} - 1$

```
struct ADQSignalProcessingParameters {
    enum ADQParameterId                         id;
    int32_t                                     reserved;
    struct ADQDigitalGainAndOffsetParameters    gain_offset;
    struct ADQSampleSkipParameters              sample_skip;
    struct ADQDbsParameters                     dbs;
    struct ADQPdrxParameters                    pdrx;
    struct ADQFirFilterParameters               fir_filter;
    struct ADQAtdParameters                     atd;
    struct ADQPdParameters                      pd;
    uint64_t                                    magic;
}
```

Description

This is a high-level struct collecting the parameters of *all* the signal processing modules of the digitizer. This means that each member struct is *also* an object that may interact with the configuration functions (see Section A.4.3). Refer to Section 5 for a high-level description of the various function modules.

Members

`id` (`enum ADQParameterId`)

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_SIGNAL_PROCESSING`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved` (`int32_t`)

Reserved

`gain_offset` (`struct ADQDigitalGainAndOffsetParameters`)

A `ADQDigitalGainAndOffsetParameters` struct representing the parameters of the digital gain and offset module (see Section 5.1).

`sample_skip` (`struct ADQSampleSkipParameters`)

A `ADQSampleSkipParameters` struct representing the parameters of the sample skip module (see Section 5.2).

`dbs` (`struct ADQDbsParameters`)

A `ADQDbsParameters` struct representing the parameters of the digital baseline stabilization module (see Section 5.3).

`pdrx` (`struct ADQPdrxParameters`)

A `ADQPdrxParameters` struct representing the parameters of the pulse detection range extension feature (see Section 5.5).

`fir_filter` (`struct ADQFirFilterParameters`)

A `ADQFirFilterParameters` struct representing the parameters of the FIR filter module (see Sec-

tion 5.4).

`atd (struct ADQAtdParameters)`

A `ADQAtdParameters` struct representing the parameters of the ATD signal processing module (see Section 5.6).

`pd (struct ADQPdParameters)`

A `ADQAtdParameters` struct representing the parameters of the PD signal processing module (see Section 5.7).

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQTestPatternParameters {
    enum ADQParameterId                     id;
    int32_t                                 reserved;
    struct ADQTestPatternParametersChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint64_t                                magic;
}
```

Description

This struct defines the parameters of the test pattern module for all channels of the digitizer. See Section 11 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_TEST_PATTERN`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved (int32_t)`

Reserved

`channel[ADQ_MAX_NOF_CHANNELS] (struct ADQTestPatternParametersChannel)`

An array of `ADQTestPatternParametersChannel` structs where each element represents the parameters for a channel. The struct at index zero targets the first channel. The constant parameter `nof_channels` holds the number of valid entries.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to `ADQ_PARAMETERS_MAGIC`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

```
struct ADQTestPatternParametersChannel {
    enum ADQTestPatternSource             source;
    int32_t                               reserved;
    struct ADQTestPatternParametersPulse  pulse;
}
```

Description

This struct is a member of `ADQTestPatternParameters` and defines test pattern parameters for a channel.

Members

`source` (`enum ADQTestPatternSource`)

The test pattern source as a value from the enumeration `ADQTestPatternSource`. The default value is `ADQ_TEST_PATTERN_SOURCE_DISABLE` which implies ADC data.

`reserved` (`int32_t`)

Reserved

`pulse` (`struct ADQTestPatternParametersPulse`)

A `ADQTestPatternParametersPulse` struct representing the parameters for the test pattern pulse generator. This parameter is only used when the `source` is set to one of the pulse generator modes. Otherwise, it is ignored.

```
struct ADQTestPatternParametersPulse {
    int64_t  baseline;
    int64_t  amplitude;
    int64_t  period;
    int64_t  width;
    int64_t  nof_pulses_in_burst;
    int64_t  nof_bursts;
    int64_t  burst_period;
    int64_t  prbs_amplitude_seed;
    int64_t  prbs_amplitude_scale;
    int64_t  prbs_width_seed;
    int64_t  prbs_width_scale;
    int64_t  prbs_noise_seed;
    int64_t  prbs_noise_scale;
    int32_t  trigger_mode_enabled;
    int32_t  reserved;
}
```

Description

This struct is a member of `ADQTestPatternParametersChannel` and defines the parameters for the test pattern pulse generator. This feature is not yet supported.

```
struct ADQTimestampSynchronizationParameters {
    enum ADQParameterId                    id;
    enum ADQEventSource                    source;
    enum ADQEdge                           edge;
    enum ADQTimestampSynchronizationMode   mode;
    enum ADQArm                            arm;
    int32_t                                reserved;
    uint64_t                               seed;
    uint64_t                               magic;
}
```

Description

This struct defines the parameters for the timestamp synchronization. See Section 7.3 for a high-level description.

Members

`id (enum ADQParameterId)`

The struct identification number. This value should always be set to `ADQ_PARAMETER_ID_TIMESTAMP_SYNCHRONIZATION`. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`source (enum ADQEventSource)`

The event source used for timestamp synchronization. Valid values are:

- `ADQ_EVENT_SOURCE_SOFTWARE`
- `ADQ_EVENT_SOURCE_TRIG`
- `ADQ_EVENT_SOURCE_PERIODIC`
- `ADQ_EVENT_SOURCE_SYNC`
- `ADQ_EVENT_SOURCE_GPIOA0`
- `ADQ_EVENT_SOURCE_GPIOB0`
- `ADQ_EVENT_SOURCE_PXIE_STARB`
- `ADQ_EVENT_SOURCE_PXIE_TRIG0`
- `ADQ_EVENT_SOURCE_PXIE_TRIG1`
- `ADQ_EVENT_SOURCE_REFERENCE_CLOCK`
- `ADQ_EVENT_SOURCE_MATRIX`

The default value is `ADQ_EVENT_SOURCE_INVALID`.

`edge (enum ADQEdge)`

An `ADQEdge` which specifies the edge sensitivity of the `source`. The default value is `ADQ_EDGE_RISING`.

`mode (enum ADQTimestampSynchronizationMode)`

Selects the timestamp synchronization mode. Valid values are:

- `ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_DISABLE`

- ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_FIRST
- ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_ALL

The default value is ADQ_TIMESTAMP_SYNCHRONIZATION_MODE_DISABLE.

`arm (enum ADQArm)`

Specifies when the timestamp synchroniztion should be armed and ready to react to events from the selected event `source`. Valid values are:

- ADQ_ARM_IMMEDIATELY
- ADQ_ARM_AT_ACQUISITION_START

The default value is ADQ_ARM_IMMEDIATELY.

`reserved (int32_t)`

reserved

`seed (uint64_t)`

Sets the seed value for the timestamp synchronization. Following a synchronization event, the timestamp is set to this value. The unit is $1/16$ of the sampling period. The default value is 0.

`magic (uint64_t)`

A magic number to indicate the end of the parameter struct. This value should always be set to ADQ_PARAMETERS_MAGIC. This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

A.3.3 Status

This section lists the structures used to communicate information about the status of the digitizer. See Section 15 for a description of the context in which these objects are used.

```
struct ADQAcquisitionStatus {
    int64_t  acquired_records[ADQ_MAX_NOF_CHANNELS];
}
```

Description

This structure defines the acqusition status. See `GetStatus()`.

Members

`acquired_records[ADQ_MAX_NOF_CHANNELS]` (`int64_t`)

The number of acquired records per channel. The record counter will wrap around to zero after 4294967295 records ($2^{32} - 1$).

```
struct ADQDataReadoutStatus {
    uint32_t  flags;
}
```

Description

This struct holds status information about a record buffer and the health of the transfer process for a specific channel. It is the type of the output parameter `status` in the function `WaitForRecordBuffer()`.

Members

`flags` (`uint32_t`)

This member is a 32-bit wide bit field holding status flags. An "all clear" status is represented by all bits being cleared (set to zero). `ADQ_DATA_READOUT_STATUS_FLAGS_OK` is an alias for this value.

`Bit 0` `ADQ_DATA_READOUT_STATUS_FLAGS_STARVING`

The channel is starved for memory. There is not a sufficient amount of buffers in circulation, causing incoming data to (eventually) be discarded, see Section 10.6 for more information.

`Bit 1` `ADQ_DATA_READOUT_STATUS_FLAGS_INCOMPLETE`

The record is incomplete. Its `header` will be *invalid* and set to `NULL`. This value is only possible if the parameter `incomplete_records_enabled` is set to a nonzero value. See Section 10.5.7 for details.

`Bit 2` `ADQ_DATA_READOUT_STATUS_FLAGS_DISCARDED`

A record was discarded due to incomplete data. This is either caused by

- an overflow of the device-to-host interface (Section 10.6); or
- an overflow of the record buffer due to its size reaching `record_buffer_size_max`.

This value is only possible if `incomplete_records_enabled` is zero (default).

```
struct ADQDramStatus {
    uint64_t  fill;
    uint64_t  fill_max;
}
```

Description

This struct defines the DRAM status. The DRAM status is sampled when `GetStatus()` is called.

Members

`fill` (`uint64_t`)
: The current fill level of the DRAM in bytes.

`fill_max` (`uint64_t`)
: The current maximum fill level of the DRAM in bytes. The maximum value is reset when `StartDataAcquisition()` is called.

```
struct ADQOverflowStatus {
    int32_t  overflow;
    int32_t  reserved;
}
```

Description

This stucture defines the overflow status. See `GetStatus()`.

Members

`overflow` (`int32_t`)
: DRAM overflow if nonzero.

`reserved` (`int32_t`)
: Reserved

```
struct ADQP2pStatus {
    struct ADQP2pStatusChannel  channel[ADQ_MAX_NOF_CHANNELS];
    uint32_t                    flags;
    int32_t                     reserved;
}
```

Description

This struct holds status information about the peer-to-peer transfer process. It is the type of the output parameter `status` in the function `WaitForP2pBuffers()`.

Members

`channel[ADQ_MAX_NOF_CHANNELS]` (`struct ADQP2pStatusChannel`)

An array of `ADQP2pStatusChannel` structs where each element represent the peer-to-peer transfer status of a channel.

`flags` (`uint32_t`)

Reserved

`reserved` (`int32_t`)

Reserved

```
struct ADQP2pStatusChannel {
    uint32_t  flags;
    int32_t   nof_completed_buffers;
    int16_t   completed_buffers[ADQ_MAX_NOF_BUFFERS];
}
```

Description

This struct is a member of `ADQP2pStatus` and holds the status information about the peer-to-peer transfer process of a channel. Refer to the documentation for `WaitForP2pBuffers()` for additional details.

Members

`flags` (`uint32_t`)

Reserved

`nof_completed_buffers` (`int32_t`)

The number of completed buffers. This parameter specifies the number of valid entries in the array `completed_buffers`.

`completed_buffers[ADQ_MAX_NOF_BUFFERS]` (`int16_t`)

An array of buffer indexes indicating which buffers hold data available for reading. The number of valid entries is specified by `nof_completed_buffers`, e.g. four completed buffers indicates that entries 0, 1, 2 and 3 each holds the index of one of the four completed buffers.

```
struct ADQTemperatureStatus {
    int32_t                          nof_sensors;
    int32_t                          reserved;
    struct ADQTemperatureStatusSensor  sensor[ADQ_MAX_NOF_TEMPERATURE_SENSORS];
}
```

Description

This struct contains the status of the digitizer's temperature sensors. See `GetStatus()`.

Members

`nof_sensors` (`int32_t`)

The number of valid entries in `sensor`.

`reserved` (`int32_t`)

Reserved

`sensor[ADQ_MAX_NOF_TEMPERATURE_SENSORS]` (`struct ADQTemperatureStatusSensor`)

An array of `ADQTemperatureStatusSensor` structs where each element represents a temperature sensor on the digitizer. The constant parameter `nof_sensors` holds the number of valid entries.

```
struct ADQTemperatureStatusSensor {
    char   label[32];
    float  value;
}
```

Description

This struct is a member of `ADQTemperatureStatus` and holds the status of a single temperature sensor.

Members

`label[32]` (`char`)

The label of the temperature sensor as a zero-terminated array of ASCII characters, i.e. a C-string.

`value` (`float`)

The current temperature of the sensor, in degrees Celsius.

```
struct ADQClockSystemStatus {
    int32_t                         nof_plls;
    int32_t                         reserved;
    struct ADQClockSystemStatusPll  pll[ADQ_MAX_NOF_PLLS];
    double                          reference_source_frequency_estimate;
}
```

Description

This struct contains the status of the digitizer's clock system. See `GetStatus()`.

Members

`nof_plls` (`int32_t`)

The number of valid entries in `pll`.

`reserved` (`int32_t`)

Reserved

`pll[ADQ_MAX_NOF_PLLS]` (`struct ADQClockSystemStatusPll`)

An array of `ADQClockSystemStatusPll` structs where each element represents a PLL in the digitizer's clock system. The constant parameter `nof_plls` holds the number of valid entries.

`reference_source_frequency_estimate` (`double`)

An estimate of the frequency of the currently selected `reference_source`, in units of Hz. If a reference source is not used in the current clock system configuration, or if the configuration does not support measuring the reference source frequency, the value will be set to -1.0.

```
struct ADQClockSystemStatusPll {
    int32_t  lock_detect;
    int32_t  lock_lost_alarm;
}
```

Description

This struct is a member of `ADQClockSystemStatus` and holds the status of a single PLL in the digitizer's clock system.

Members

`lock_detect` (`int32_t`)

The lock detection status of the PLL, where

- 1 indicates that the PLL is currently locked,
- 0 indicates that the PLL is unlocked; and
- -1 indicates that the PLL is not used in the current clock system configuration.

`lock_lost_alarm` (`int32_t`)

An alarm state for the PLL where

- 1 that the PLL has lost its lock at some point since the alarm was last cleared,
- 0 indicates that no loss of lock has occured; and
- -1 indicates that the PLL is not used in the current clock system configuration, or that the PLL does not support the alarm feature.

The alarm is cleared after each read of `ADQClockSystemStatus` via `GetStatus()`, and also after each reconfiguration of the `ADQClockSystemParameters`.

```
struct ADQTimestampSynchronizationStatus {
    int32_t  counter;
    int32_t  reserved;
}
```

Description

This structure defines the timestamp synchronization status. See `GetStatus()`.

Members

`counter (int32_t)`

The number of times the timestamp has been synchronized. The value is only valid if the timestamp synchronization mechanism (see Section 7.3) is active.

`reserved (int32_t)`

Reserved

```
struct ADQDaisyChainStatus {
    int32_t  setup_time_warning;
    int32_t  rearm_error;
}
```

Description

This structure defines the daisy chain status. See `GetStatus()` and Section 7.4.8 for more information.

Members

`setup_time_warning (int32_t)`

This value is asserted if a secondary digitizer has received an edge of the daisy chain signal close to the edge of the refrence clock and zero otherwise. The value is sticky and is reset when read.

Important

This value is only valid for secondary digitizers.

`rearm_error (int32_t)`

This value is asserted if the primary digitizer's daisy chain signal propagation module has detected a trigger event during its rearm period and zero otherwise. The value is sticky and is reset when read.

Important

This value is only valid for the primary digitizer.

```
struct ADQLicenseStatus {
    int32_t  valid;
    int32_t  reserved;
}
```

Description

This struct contains the status of the digitizer's license information. See `GetStatus()`.

Members

`valid` (`int32_t`)

A boolean value which indicates whether or not the licenses stored in the digitizer's nonvolatile memory is valid (and sufficient) to run the active firmware.

- When the value is 1, the available licenses fulfill the requirements of the active firmware.
- When the value is 0, the available licenses do not fulfill the requirements of the active firmware.

Important

Acquiring data will not be possible when this value is 0.

`reserved` (`int32_t`)

Reserved

A.3.4 Data

This section lists the structures used to represent the data transferred by the digitizer. See Section 15 for a description of the context in which these objects are used.

```
struct ADQGen4Record {
    struct ADQGen4RecordHeader * header;
    void                       * data;
    uint64_t                     size;
}
```

Description

This struct defines the expected memory format of a *record buffer* rotating in the `WaitForRecordBuffer()` / `ReturnRecordBuffer()` interface.

Members

`header` (`struct ADQGen4RecordHeader *`)

A pointer to an `ADQGen4RecordHeader`.

> **Important**
>
> This member only points to valid memory if metadata is enabled, i.e. the data transfer parameter `metadata_enabled` is set to a nonzero value. Otherwise, it is set to `NULL`. Make sure to not access this member if metadata is disabled. It is also set to `NULL` for an incomplete record (Section 10.5.7), which is indicated by the data readout `status flags` signaling `ADQ_DATA_READOUT_STATUS_FLAGS_INCOMPLETE`.

`data` (`void *`)

A pointer to a memory region holding the record data. The member `size` *must* be set to the size of this region. This is handled automatically if the API is tasked with managing the memory (`memory_owner` is set to `ADQ_MEMORY_OWNER_API`). In that case it is instead important to *never* change the value.

The header field `data_format` should be used to interpret the memory region. For example, if the value is `ADQ_DATA_FORMAT_INT16`, the region contains 16-bit signed integer values and should be traversed using a matching pointer type.

> **Important**
>
> When manually allocating record buffers, make sure to set the value of `size` to the size of the memory region pointed to by `data`.

`size` (`uint64_t`)

The size (in bytes) of the memory region pointed to by `data`. This is *not* the amount of data available for reading, but rather the *capacity* of the record buffer. The number of bytes available for reading is returned by `WaitForRecordBuffer()`.

```
struct ADQGen4RecordHeader {
    uint8_t   version_major;
    uint8_t   version_minor;
    uint16_t  timestamp_synchronization_counter;
    uint16_t  general_purpose_start;
    uint16_t  general_purpose_stop;
    uint64_t  timestamp;
    int64_t   record_start;
    uint32_t  record_length;
    uint8_t   user_id;
    uint8_t   misc;
    uint16_t  record_status;
    uint32_t  record_number;
    uint8_t   channel;
    uint8_t   data_format;
    char      serial_number[10];
    uint64_t  sampling_period;
    double    time_unit;
    uint32_t  firmware_specific;
    int32_t   reserved;
}
```

Description

The header structure contained in an `ADQGen4Record`. Some members require post processing to be valid. This processing is not available when the data transfer interface (Section 10.4) is used. Such members are marked "✖ Data transfer". The absence of any symbol implies that the member is valid for both the data readout and data transfer processes.

Note

This definition is valid for header version `2.0`.

Members

`version_major` (`uint8_t`) ✖ Data transfer

The major version number, i.e. `2` in `2.0`. The full version number denotes *binary compatibility* between definitions of this data type and follows these rules:

- Two headers with different *major* version number *cannot* be parsed using the same definition. The major version number is incremented on a size change, or if any existing members change their name or type.
- Two headers with different *minor* version number can be *partially* understood using the *lower* definition. It is implied that the headers are of the same size, and that the existing (valid) members are the same. There are few circumstances when the minor version number is incremented, but one such case is when a *reserved* field is implemented.

`version_minor` (`uint8_t`) ✖ Data transfer

The minor version number, i.e. `0` in `2.0`. See `version_major` for additional details.

`timestamp_synchronization_counter` (`uint16_t`)

If the timestamp synchronization mechanism (see Section 7.3) is active, this field will hold the number of times that the timestamp had been synchronized when the record was acquired.

`general_purpose_start` (`uint16_t`)

Reserved

`general_purpose_stop` (`uint16_t`)

Reserved

`timestamp` (`uint64_t`)

The timestamp of the trigger event, expressed in time units (`time_unit`).

`record_start` (`int64_t`) ✖ Data transfer

The time between the trigger event and the first sample in the record, expressed in time units (`time_unit`). This means that the timestamp of the first sample in the record is *the sum* of the values of `timestamp` and `record_start`. See (27) and Section 9.3 for more information. Only valid when the data readout interface is used (see Section 10.5).

- A value less than zero implies that the first sample in the record was acquired *before* the trigger event occurred (pretrigger).
- A value equal to zero implies that the first sample in the record was acquired *precisely* when the trigger event occurred.
- A value greater than zero implies that the first sample in the record was acquired *after* the trigger event occurred (trigger delay).

`record_length` (`uint32_t`)

The length of the record, expressed in *samples*.

`user_id` (`uint8_t`)

An 8-bit value that may be set from the development kit.

`misc` (`uint8_t`)

A bit field containing miscellanous information:

Bits 7–4:

Reserved

Bits 3–0:

The state (logic level) of the pattern generator outputs (Section 7.1) at the time when the record was acquired. Each pattern generator claims one bit in the range with the state of `ADQ_FUNCTION_PATTERN_GENERATOR0` at bit 0 and so on.

`record_status` (`uint16_t`)

A bit field containing assorted information about the record itself, or the digitizer at the time of acquisition.

Bits 15-4:

Reserved

Bit 3:

This bit is asserted if the record was triggered by a rising edge event (`ADQ_EDGE_RISING`) from the selected `trigger_source`. This is useful to differentiate between records when the `trigger_edge` is set to `ADQ_EDGE_BOTH`.

To check for a rising edge trigger condition, mask `record_status` with `ADQ_RECORD_STATUS_RISING_EDGE` and test for a nonzero value.

Bit 2:

This bit is asserted if one or several samples in the record have saturated at the maximum or minimum value when their actual values cannot be represented in the available range.

To check for an overrange condition, mask `record_status` with `ADQ_RECORD_STATUS_OVERRANGE` and test for a nonzero value.

Note

On FWATD (Section 5.6) this bit indicates an arithmetic overflow, i.e. that the accumulation process was forced to saturate the data. Thus, it indicates that (at least) one sample in an accumulation result record has saturated, and not that a record with an overrange condition has been accumulated.

Bit 1:

Reserved

Bit 0:

This bit is asserted when the record is incomplete and has lost data at the end. The data readout interface discards these records by default (Section 10.5.6) but this behavior can be changed by allowing incomplete records to propagate (Section 10.5.7). Records with missing data are the result of an overflow condition when the digitizer has been configured to keep the acquisition going, rather than to stop (the default behavior). See Section 10.6.3 for additional details.

To check for this condition condition, mask `record_status` with `ADQ_RECORD_STATUS_OVERFLOW` and test for a nonzero value.

`record_number` (`uint32_t`)

The record number as a 32-bit unsigned value. The first record acquired after `StartDataAcquisition()` will have this field set to zero. When the data transfer interface is used (see Section 10.4) the value will be limited to 16 bits.

Important

The record number wraps to zero at the maximum value.

`channel (uint8_t)` ✖ Data transfer

The channel from which the record originated.

`data_format (uint8_t)` ✖ Data transfer

The format of the record `data`.

`ADQ_DATA_FORMAT_INT16`

The record `data` consists of samples with a 16-bit 2's complement representation.

`ADQ_DATA_FORMAT_INT32`

The record `data` consists of samples with a 32-bit 2's complement representation.

`ADQ_DATA_FORMAT_PULSE_ATTRIBUTES`

The record `data` consists of `ADQPulseAttributes` objects. Records with this data format are only generated by the FWPD firmware, see Section 5.7.

`serial_number[10] (char)` ✖ Data transfer

The digitizer's serial number as a zero-terminated array of ASCII characters, i.e. a C-string. For example, `"SPD-09999"`.

`sampling_period (uint64_t)` ✖ Data transfer

The time between two samples, expressed in time units (`time_unit`).

`time_unit (double)` ✖ Data transfer

The value of a *time unit* in seconds. The header fields `timestamp`, `record_start` and `sampling_period` are integer values which may be converted to seconds by multiplying with the time unit.

`firmware_specific (uint32_t)` ✖ Data transfer

A 32-bit value whose contents is specific to the current firmware (Section 1.3).

- `FWDAQ`: Reserved, reads as zero.
- `FWPD`: Reserved, reads as zero.
- `FWATD`: An unsigned 32-bit value indicating the number of accumulations in the record. This value should be used to normalize the data when converting to a voltage. Refer to Section 5.6.2 and (7).
  The value can differ from the value of `nof_accumulations` due to the FWATD overflow mechanism (Section 5.6.6).

`reserved (int32_t)` ✖ Data transfer

Reserved

```
struct ADQGen4RecordArray {
    struct ADQGen4Record ** record;
    int32_t                nof_records;
}
```

Description

This struct defines the expected memory format of a *record buffer array* rotating in the `WaitForRecordBuffer()` / `ReturnRecordBuffer()` interface when the parameter `nof_record_buffers_in_array` is set to a nonzero value.

Members

`record` (`struct ADQGen4Record **`)

A pointer to the start of an array of pointers where each element points to an `ADQGen4Record` object.

`nof_records` (`int32_t`)

The number of valid records in the `record` array.

```
struct ADQPulseAttributes {
    int32_t   area;
    uint32_t  peak_position;
    uint16_t  peak;
    uint16_t  fwhm;
    uint8_t   status;
    uint8_t   reserved[3];
}
```

Description

This struct defines the memory format for pulse attributes extracted by the PD signal processing module (exclusive to the FWPD firmware). A record's `data` region contains objects of this type when its header field `data_format` is set to `ADQ_DATA_FORMAT_PULSE_ATTRIBUTES`. See Section 5.7 for additional details.

Members

`area` (`int32_t`)

The area of the pulse measured in ADC codes relative to the `baseline`. The configured `polarity` affects how this area is calculated. Negative values are possible. Refer to Section 5.7.7 for more information.

`peak_position` (`uint32_t`)

The position of the `peak` value relative to the first sample in the record, given as an offset measured in samples. Where the `peak` is located on the digitizer's timing grid (Section 9.3) can be expressed as

$$\texttt{timestamp} + \texttt{record\_start} + \texttt{peak\_position} \cdot \texttt{sampling\_period}.$$

Refer to Section 5.7.5 for more information.

`peak` (`uint16_t`)

The absolute value of the peak (extreme value) relative to the `baseline`. Refer to Section 5.7.5 for more information.

`fwhm` (`uint16_t`)

The full width at half maximum, as defined by the `peak` value. Refer to Section 5.7.6 for more information.

`status` (`uint8_t`)

A bit field containing status information about the attributes.

Bits 7-1:

Reserved

Bit 0:

This bit is asserted if *all* of the attributes are valid. If the bit is zero, one or several attributes are invalid and cannot be trusted. To check for valid attribute data, mask `status` with `ADQ_PULSE_ATTRIBUTES_STATUS_VALID` and test for a nonzero value. See Section 5.7.4 for more information.

`reserved[3]` (`uint8_t`)

Reserved

A.3.5 Other

This section lists structures used for setting up and managing the digitizer.

```
struct ADQInfoListEntry {
    enum ADQHWIFEnum        HWIFType;
    enum ADQProductID_Enum  ProductID;
    unsigned int            VendorID;
    unsigned int            AddressField1;
    unsigned int            AddressField2;
    char[64]                DevFile;
    unsigned int            DeviceInterfaceOpened;
    unsigned int            DeviceSetupCompleted;
}
```

Description

This struct defines the device information entry of the `adq_info_list` returned by `ADQControlUnit_ListDevices()`.

A.4 Functions

This section lists the data structures used when configuring and controlling the digitizer. These are defined in the ADQAPI header file `ADQAPI.h` and versioned by the two constants `ADQAPI_VERSION_MAJOR` and `ADQAPI_VERSION_MAJOR`. See `ADQAPI_ValidateVersion()` for more information about how to implement version control in the user application space.

A.4.1 General


```
int ADQAPI_ValidateVersion(
    int  major,
    int  minor
)
```

Validate the version used by the user application and the API.

Return value

This function returns 0 if the API version is compatible, -1 if it is incompatible and -2 if it is backwards compatible.

Description

This function provides a safe-guarding mechanism against dynamically linking a precompiled version of the user application against an incompatible API. The protection works by adding a call to this function with the static arguments ADQAPI_VERSION_MAJOR and ADQAPI_VERSION_MINOR:

```
int result = ADQAPI_ValidateVersion(ADQAPI_VERSION_MAJOR, ADQAPI_VERSION_MINOR);
```

This version number is defined as a constant in the ADQAPI.h header file. The result is a handshake between the user application and the API evaluated at runtime—allowing the user to take appropriate action, rather than to experience errors that are potentially hard to find.

Parameters

major (int)
: The major version number. Should always be set to ADQAPI_VERSION_MAJOR.

minor (int)
: The minor version number. Should always be set to ADQAPI_VERSION_MINOR.

A.4.2 Identification

```
void * CreateADQControlUnit()
```

Creates the ADQ control unit.

Return value

A pointer to the control unit object.

Description

This function creates an instance of the ADQ control unit, that may be used to find and setup ADQ devices. This function should only be called once.

```
int ADQControlUnit_EnableErrorTrace(
    void         * adq_cu,
    unsigned int   trace_level,
    const char   * trace_file_dir
)
```

Enables message logging to file.

Return value

Returns 1 if the operation is successful, otherwise 0.

Description

Calling this function enables logging for the control unit and all connected devices.

Parameters

`adq_cu (void *)`

Pointer to the control unit instance, created by `CreateADQControlUnit()`.

`trace_level (unsigned int)`

Selects the logging level. The following levels are supported:

- `LOG_LEVEL_ERROR`: Error
- `LOG_LEVEL_WARN`: Error and warning
- `LOG_LEVEL_INFO`: Error, warning and information

Setting bit 11 of this argument enables timestamps.

`trace_file_dir (const char *)`

Either a path to a directory or a path to a file. If a file path is specified all log messages will be appended to this file. If a directory path is specified the control unit messages will be logged to

```
<trace_file_dir>/spd_adqcontrolunit_trace.log
```

and a sperate file for each device will be created on the format

```
<trace_file_dir>/spd_device_<ADQ Type>_<Hardware Address>_trace.log
```

The current working directory can be specified as `"."`.

```
int ADQControlUnit_ListDevices(
    void                     * adq_cu,
    struct ADQInfoListEntry ** adq_info_list,
    unsigned int             * adq_info_list_length
)
```

List avilable devices

Return value

Returns 1 if the operation is successful, otherwise 0.

Description

This function lists available devices without setting up a communication channel.

Parameters

`adq_cu (void *)`

Pointer to the control unit instance, created by `CreateADQControlUnit()`.

`adq_info_list (struct ADQInfoListEntry **)`

Pointer to a `ADQInfoListEntry` pointer. The API will allocate the memory and populate it with one `ADQInfoListEntry` per device.

`adq_info_list_length (unsigned int *)`

The number of entries in `adq_info_list`.

```
int ADQControlUnit_SetupDevice(
    void * adq_cu,
    int    adq_info_list_entry_number
)
```

Set up the device

Return value

Returns 1 if the operation is successful, otherwise 0.

Description

This function is called after `ADQControlUnit_ListDevices()` to set up the device, see Section 15.3.

Parameters

`adq_cu (void *)`

Pointer to the control unit instance, created by `CreateADQControlUnit()`.

`adq_info_list_entry_number (int)`

The index of the device to set up, starting at 0.

A.4.3 Parameter Interface

```
int InitializeParameters(
    enum ADQParameterId  id,
    void *const          parameters
)
```

Initialize a parameter set to its default values.

Return value

If the operation is successful, the return value is set to the size of the initialized parameter set. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`InitializeParametersString()`, `InitializeParametersFilename()`

Description

This function initializes the memory region pointed to by `parameters` to hold the default values of the parameter set `id`. Refer to the parameter definitions in Section A.3 for information on the default values for each parameter set. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`parameters (void *const)`

A pointer to a memory region of sufficient size to accommodate the target parameter set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int InitializeParametersString(
    enum ADQParameterId  id,
    char *const          string,
    size_t               length,
    int                  format
)
```

Initialize a parameter set to its default values and store the result encoded as JSON in a zero terminated array of ASCII characters (C-string).

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the string buffer. This includes the zero terminator. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`InitializeParameters()`, `InitializeParametersFilename()`

Description

This function is similar to `InitializeParameters()`, except that the parameter set is encoded as JSON and written to the string buffer pointed to by `string`. The `length`, i.e. capacity, of the string buffer needs to be sufficiently large to receive the encoded parameter set. If the required length is larger than this value, the operation fails with `ADQ_EINVAL`. If `format` is set to a nonzero value, formatted JSON will be written to the string buffer. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`string (char *const)`

A pointer to a string buffer of sufficient size to accommodate the target parameter set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`length (size_t)`

The length (capacity) of the `string` in bytes. This length includes the zero terminator. If the required length is larger than this value, the function fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the string buffer.

```
int InitializeParametersFilename(
    enum ADQParameterId  id,
    const char *const    filename,
    int                  format
)
```

Initialize a parameter set to its default values and store the result encoded as JSON in a target file.

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the file. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`InitializeParameters()`, `InitializeParametersString()`

Description

This function is similar to `InitializeParametersString()`, except that the parameter set is written to the file `filename` instead of a string buffer. An existing file is overwritten by this operation. If the file cannot be opened for writing or a low level I/O operation fails, `ADQ_EEXTERNAL` is returned. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`filename (const char *const)`

A pointer to a zero terminated array of ASCII characters (C-string) that holds the system path to the target file. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the target file.

```
int GetParameters(
    enum ADQParameterId  id,
    void *const          parameters
)
```

Read the current values of a parameter set from the digitizer.

Return value

If the operation is successful, the return value is set to the size of the retrieved parameter set. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetParametersString()`, `GetParametersFilename()`

Description

This function reads the current values of the parameter set `id` into the memory region pointed to by `parameters`. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`parameters (void *const)`

A pointer to a memory region of sufficient size to accommodate the target parameter set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int GetParametersString(
    enum ADQParameterId  id,
    char *const          string,
    size_t               length,
    int                  format
)
```

Read the current values of a parameter set from the digitizer and store the result encoded as JSON in a zero terminated array of ASCII characters (C-string).

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the string buffer. This includes the zero terminator. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetParameters()`, `GetParametersFilename()`

Description

This function is similar to `GetParameters()`, except that the parameter set is encoded as JSON and written to the string buffer pointed to by `string`. The `length`, i.e. capacity, of the string buffer needs to be sufficiently large to receive the encoded parameter set. If the required length is larger than this value, the operation fails with `ADQ_EINVAL`. If `format` is set to a nonzero value, formatted JSON will be written to the string buffer. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`string (char *const)`

A pointer to a string buffer of sufficient size to accommodate the target parameter set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`length (size_t)`

The length (capacity) of the `string` in bytes. This length includes the zero terminator. If the required length is larger than this value, the function fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the string buffer.

```
int GetParametersFilename(
    enum ADQParameterId  id,
    const char *const    filename,
    int                  format
)
```

Read the current values of a parameter set from the digitizer and store the result encoded as JSON in a target file.

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the file. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetParameters()`, `GetParametersString()`

Description

This function is similar to `GetParametersString()`, except that the parameter set is written to the file `filename` instead of a string buffer. An existing file is overwritten by this operation. If the file cannot be opened for writing or a low level I/O operation fails, `ADQ_EEXTERNAL` is returned. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQParameterId` in Section A.2 for more information.

`filename (const char *const)`

A pointer to a zero terminated array of ASCII characters (C-string) that holds the system path to the target file. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the target file.

```
int SetParameters(
    void *const  parameters
)
```

Validate and write a parameter set to the digitizer.

Return value

If the operation is successful, the return value is set to the size of the written parameter set. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`SetParametersString()`, `SetParametersFilename()`

Description

This function writes the parameter set pointed to by `parameters` to the digitizer. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`parameters (void *const)`

A pointer to a memory region holding the target parameter set. The identification number is read from the parameter set itself. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int SetParametersString(
    const char *const  string,
    size_t             length
)
```

Validate and write a parameter set to the digitizer where the parameters are read as JSON from a zero terminated array of ASCII characters (C-string).

Return value

If the operation is successful, the return value is set to the number of characters (bytes) read from the string buffer. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`SetParameters()`, `SetParametersFilename()`

Description

This function is similar to `SetParameters()`, except that the parameter set is read as JSON from the string buffer pointed to by `string`. The parsing continues until a valid JSON object has been constructed, or the maximum `length` has been exceeded. The latter case results in an error with `ADQ_EINVAL` as the return value. On success, it is expected that the return value may be less than `length`, depending on the contents of the string buffer. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`string (const char *const)`

A pointer to a string buffer holding the target parameter set encoded as JSON. The identification number is read from the parameter set itself. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`length (size_t)`

This value specifies the maximum number of characters that can be safely read from the string buffer. Normally, this is the length of the `string` in bytes. The parsing expects a valid JSON object to exist within the provided bounds.

```
int SetParametersFilename(
    const char *const  filename
)
```

Validate and write a parameter set to the digitizer where the parameters are read as JSON from a target file.

Return value

If the operation is successful, the return value is set to the number of characters (bytes) read from the file. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`SetParameters()`, `SetParametersString()`

Description

This function is similar to `SetParametersString()`, except that the parameter set is read from the file `filename` instead from a string buffer. The file must contain a valid JSON object starting at the first character. If the file does not exist or cannot be open for reading, the operation fails with `ADQ_EEXTERNAL`. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`filename (const char *const)`

A pointer to a zero terminated array of ASCII characters (C-string) that holds the system path to the target file. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int ValidateParameters(
    const void *const  parameters
)
```

Validate (but do not apply) a parameter set.

Return value

If the operation is successful, the return value is set to the size of the validated parameter set. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`ValidateParametersString()`, `ValidateParametersFilename()`

Description

This function validates the input parameter set according to the same rules as `SetParameters()`. However, the parameters are *not* applied. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`parameters` (`const void *const`)

A pointer to a memory region holding the target parameter set. The identification number is read from the parameter set itself. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int ValidateParametersString(
    const void *const  string,
    size_t             length
)
```

Validate (but do not apply) a parameter set read as JSON from a zero-terminated array of ASCII characters (C-string).

Return value

If the operation is successful, the return value is set to the number of characters (bytes) read from the string buffer. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`ValidateParameters()`, `ValidateParametersFilename()`

Description

This function is similar to `ValidateParameters()`, except that the parameter set is read as JSON from the string buffer pointed to by `string`. The parsing continues until a valid JSON object has been constructed, or the maximum `length` has been exceeded. The latter case results in an error with `ADQ_EINVAL` as the return value. On success, it is expected that the return value may be less than `length`, depending on

the contents of the string buffer. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`string (const void *const)`

A pointer to a string buffer holding the target parameter set encoded as JSON. The identification number is read from the parameter set itself. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`length (size_t)`

This value specifies the maximum number of characters that can be safely read from the string buffer. Normally, this is the length of the `string` in bytes. The parsing expects a valid JSON object to exist within the provided bounds.

```
int ValidateParametersFilename(
    const char *const  filename
)
```

Validate (but do not apply) a parameter set read as JSON from a target file.

Return value

If the operation is successful, the return value is set to the number of characters (bytes) read from the file. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`ValidateParameters()`, `ValidateParametersString()`

Description

This function is similar to `ValidateParameters()`, except that a JSON encoded parameter set is read from the file `filename`. The file must contain a valid JSON object starting at the first character. If the file does not exist or cannot be open for reading, the operation fails with `ADQ_EEXTERNAL`. Refer to Section 15.5 for a high-level description of the configuration interface.

Parameters

`filename (const char *const)`

A pointer to a zero terminated array of ASCII characters (C-string) that holds the system path to the target file. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

A.4.4 Data Acquisition

`int StartDataAcquisition(void)`

Start the data acquisition, data transfer and data readout processes.

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function behaves differently depending on the digitizer's configuration at the time of the call:

Data readout

If the configuration fulfills the critera in Section 10.5 this function will start the data acquisition, data transfer and data readout processes—effectively arming the digitizer. If the operation is successful, the digitizer will be under the control of the API and the user *must not* call any API functions other than those marked "🔀 Thread-safe". Calling `StopDataAcquisition()` stops the acquisition and transfer of data and returns control to the user.

> **Important**
>
> Once the data acquisition process has started, the user must not call any API fuctions other than those marked "🔀 Thread-safe".

Data transfer

If the configuration fulfills the criteria in Section 10.4 this function will start the data acquisition and data transfer processes—effectively arming the digitizer. Calling `StopDataAcquisition()` stops the acquisition and the transfer of data.

If the trigger blocking function is active for any channel (Section 9.5), that mechanism is armed together with the data acquisition process.

`int StopDataAcquisition(void)` 🔀 Thread-safe

Stop the data acquisition, data transfer and data readout processes.

Return value

If the operation is successful, `ADQ_EOK` is returned. Additionally, `ADQ_EINTERRUPTED` may be an expected return value if an acquisition is stopped prematurely. Other negative values indicate that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

Calling this function stops the data acquisition, data transfer and data readout processes in a well-defined manner. If the data readout process was running (Section 10.5), this function marks the point where control of the digitizer is returned to the user. This function *must* be called before disconnecting from the digitizer if an acquisition is running.

Important

This function frees the record buffer memory.

A.4.5 Data Transfer

```
int WaitForP2pBuffers(
    struct ADQP2pStatus * status,
    int                   timeout
)
```

Wait for data from the data transfer process.

Return value

If the operation is successful, `ADQ_EOK` is returned. The return value `ADQ_EAGAIN` indicates that the operation timed out. Other negative values indicate that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

Wait for new data to become available by observing the transfer buffer markers for all active channels. This function returns as soon as at least one transfer buffer is filled, or `timeout` is reached. This function is only used with the data transfer interface (Section 10.4) and when `marker_mode` is set to `ADQ_MARKER_MODE_HOST_MANUAL`. Refer to Section 10.4.2 for a program flowchart.

Parameters

`status (struct ADQP2pStatus *)`

The `status` parameter is a pointer to an `ADQP2pStatus` struct whose value communicates status information about the data transfer transfer process. If the function returns `ADQ_EOK`, information on which transfer buffers are available for reading is found in in this struct. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`timeout (int)`

This parameter determines the behavior when a transfer buffer is not immediately available:

- Any positive value (> 0) waits `timeout` milliseconds.
- The value 0 causes the function to return immediately.

A negative value will cause the operation to fail with `ADQ_EINVAL`.

```
int UnlockP2pBuffers(
    int       channel,
    uint64_t  mask
)
```

Unlock one or several transfer buffers for the target channel.

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

Calling this function will unlock one or several transfer buffers for the target channel. Once a transfer buffer is unlocked, its contents can be overwritten by the digitizer at any time. This function is only used with the data transfer interface (Section 10.4) and when `write_lock_enabled` is set to 1.

Parameters

`channel (int)`

Index of the target channel.

`mask (uint64_t)`

A mask of buffer indexes to unlock. Each bit position in the mask corresponds to a buffer index. For example, set the mask to `0x30` to unlock buffer 4 and 5.

A.4.6 Data Readout

```
int64_t WaitForRecordBuffer(                                    Thread-safe
    int                          * channel,
    void                         ** buffer,
    int                            timeout,
    struct ADQDataReadoutStatus  * status
)
```

Wait for data from the target channel.

Return value

If the operation is successful, the return value depends on the parameter `nof_record_buffers_in_array`. The return value holds either

1. the size of the record buffer's data payload in bytes, if the parameter is set to zero; or
2. the number of record buffers in the returned array, if the parameter is set to a nonzero value.

Note

Case 1 defines the default behavior. Case 2 is tied to an advanced use case, see Section 10.5.8 for more information.

The return value `ADQ_EAGAIN` indicates that the operation timed out. Other negative values indicate that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

Note

This function is used with the data readout interface (Section 10.5).

`WaitForRecordBuffer()` allows access to the read port of a channel. Through this port, a channel can pass record buffers, status information or both. Currently, the `status` parameter is unused by ADQ3 series digitizers. This will change in future releases.

It is important to note that writing data to a record buffer is not triggered by a call to this function. Instead, this happens continuously in the background and is managed by the internal thread (Fig. 45). The function notifies the user of the location of a completed record buffer by passing a reference via the parameter `buffer`. This action does *not* transfer ownership of the memory. The memory of the underlying record buffer is owned by the API.

Though the memory is owned by the API, once a reference to a record buffer is passed to the user application, the API will not attempt to access the underlying memory for any reason. To make the memory available to the API once again, the user has to call `ReturnRecordBuffer()`. These two functions work in tandem to achieve an efficient memory utilization suitable to transfer data indefinitely.

Parameters

`channel (int *)`

The `channel` parameter is a pointer to an `int` whose value specifies for which channel to wait for a record buffer. The indexing is zero based, i.e. the value 0 corresponds to the first channel.

The channel index is passed by pointer and by value to handle the special value `ADQ_ANY_CHANNEL`. In this case, the operation will return as soon as a record buffer can be read from any of the active channels (or an error occurs). If the operation is successful, the API will set the value pointed to by `channel` to the channel that responded. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`buffer (void **)`

The `buffer` parameter is a pointer to a `void*` whose value indicates where the record buffer is located. In other words, this parameter *outputs* an address to a memory region where data is available for reading. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

When this interface is used by an ADQ3 series digitizer, the buffer points to an `ADQGen4Record` struct if the parameter `nof_record_buffers_in_array` is set to zero (the default). Otherwise, the buffer points to an `ADQGen4RecordArray` which holds one or several record buffers. See Section 10.5.8 for more information.

```
/* Declare a pointer to receive the location of a record buffer. */
struct ADQGen4Record *record = NULL;

/* Request data from the first channel with a timeout of 1000 milliseconds. */
int64_t result = WaitForRecordBuffer(0, &record, 1000, NULL);
```

`timeout (int)`

This parameter determines the behavior when a record buffer is not immediately available:

- Any positive value (> 0) waits `timeout` milliseconds.
- The value 0 causes the function to return immediately.
- The value -1 causes the function to wait indefinitely.

`status (struct ADQDataReadoutStatus *)`

The `status` parameter is a pointer to an `ADQDataReadoutStatus` struct whose value communicates status information about the record buffer and the health of the transfer process for the target channel. The value `NULL` is allowed and prevents the propagation of status information. See Section 10.5.4 for more information.

```
int ReturnRecordBuffer(
    int    channel,
    void * buffer
)
```

Thread-safe

Return memory to be used by the data readout process.

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

Note

This function is used with the data readout interface (Section 10.5).

`ReturnRecordBuffer()` allows access to the write port of a channel. Through this port, the user passes references to memory regions to be used to store the data associated with a record.

Parameters

`channel (int)`

The `channel` parameter specifies to which channel the record buffer is returned. The special value `ADQ_ANY_CHANNEL` is a wildcard value to task the API with finding the record buffer's corresponding channel. However, this operation requires an internal table-based lookup so it is always more efficient to specify the channel explicitly. The indexing is zero based, i.e. the value 0 corresponds to the first channel.

`buffer (void *)`

The `buffer` parameter is a pointer to a memory region that should be used to receive a new record buffer. Once the memory is handed over to the API, modification of its contents may happen at any time. If `buffer` is `NULL`, the operation fails with `ADQ_EINVAL`.

When this interface is used by an ADQ3 series digitizer, the buffer points to an `ADQGen4Record` struct if the parameter `nof_record_buffers_in_array` is set to zero (the default). Otherwise, the buffer points to an `ADQGen4RecordArray` which holds one or several record buffers. See Section 10.5.8 for more information.

Since the API owns the memory used in the data readout process (see Section 10.5.2), the value of `buffer` is expected to exactly match the values passed to the user application via `WaitForRecordBuffer()`.

A.4.7 Status Monitoring

```
int GetStatus(
    enum ADQStatusId  id,
    void *const       status
)
```

Thread-safe

Read the current status of digitizer.

Return value

If the operation is successful, the return value is set to the size of the retrieved status set. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetStatusString()`, `GetStatusFilename()`

Description

This function reads the current values of the status set `id` into the memory region pointed to by `status`.

Parameters

`id (enum ADQStatusId)`

The status set's identification number. Targeting an unsupported status set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQStatusId` in Section A.2 for more information.

`status (void *const)`

A pointer to a memory region of sufficient size to accommodate the target status set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

```
int GetStatusString(
    enum ADQStatusId  id,
    char *const       string,
    size_t            length,
    int               format
)
```

Thread-safe

Read the current status of the digitizer and store the result encoded as JSON in a zero terminated array of ASCII characters (C-string).

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the

string buffer. This includes the zero terminator. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetStatus()`, `GetStatusFilename()`

Description

This function is similar to `GetStatus()`, except that the parameter set is encoded as JSON and written to the string buffer pointed to by `string`. The `length`, i.e. capacity, of the string buffer needs to be sufficiently large to receive the encoded parameter set. If the required length is larger than this value, the operation fails with `ADQ_EINVAL`. If `format` is set to a nonzero value, formatted JSON will be written to the string buffer.

Parameters

`id (enum ADQStatusId)`

The status set's identification number. Targeting an unsupported status set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQStatusId` in Section A.2 for more information.

`string (char *const)`

A pointer to a string buffer of sufficient size to accommodate the target parameter set. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`length (size_t)`

The length (capacity) of the `string` in bytes. This length includes the zero terminator. If the required length is larger than this value, the function fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the string buffer.

```
int GetStatusFilename(
    enum ADQStatusId   id,
    const char *const  filename,
    int                format
)
```

Thread-safe

Read the current status of the digitizer and store the result encoded as JSON in a target file.

Return value

If the operation is successful, the return value is set to the number of characters (bytes) written to the string buffer. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

See also

`GetStatus()`, `GetStatusString()`

Description

This function is similar to `GetStatusString()`, except that the parameter set is written to the file `filename` instead of a string buffer. An existing file is overwritten by this operation. If the file cannot be opened for writing or a low level I/O operation fails, `ADQ_EEXTERNAL` is returned.

Parameters

`id (enum ADQStatusId)`

The status set's identification number. Targeting an unsupported status set will cause the operation to fail with `ADQ_EINVAL`. Refer to the enumeration `ADQStatusId` in Section A.2 for more information.

`filename (const char *const)`

A pointer to a zero terminated array of ASCII characters (C-string) that holds the system path to the target file. If this parameter is `NULL`, the operation fails with `ADQ_EINVAL`.

`format (int)`

If this parameter is set to a nonzero value, formatted JSON will be written to the target file.

A.4.8 Cleanup

```
void DeleteADQControlUnit(
    void * adq_cu
)
```

Deletes the control unit.

Description

This function deletes the control unit, the devices and all other resources allocated by the API.

Parameters

`adq_cu (void *)`

Pointer to the control unit object.

A.4.9 EEPROM

```
int WriteEeprom(
    enum ADQEeprom     eeprom,
    uint32_t           address,
    const void *const  buffer,
    size_t             length
)
```

Write data to the digitizer's EEPROM.

Return value

If the operation is successful, the return value is set to the number of bytes written to the digitizer's EEPROM. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function reads `length` bytes from the `buffer` and writes the data to the target EEPROM, starting at the target `address`. The user is only allowed to target the area `ADQ_EEPROM_USER`. Refer to Section 14 for more information.

Parameters

`eeprom (enum ADQEeprom)`

The target EEPROM area as a value from the enumeration `ADQEeprom`. Only the area `ADQ_EEPROM_USER` is available to the user. Writing to any other area will cause the function to fail with `ADQ_EUNSUPPORTED` as the return value.

`address (uint32_t)`

The target address within the EEPROM area. The data in the `buffer` will be placed starting at this point.

`buffer (const void *const)`

A pointer to a memory region of (at least) size `length` containing the data to be written to the EEPROM.

`length (size_t)`

The number of bytes to read from `buffer` and write to the EEPROM. It is implied that the memory region pointed to by `buffer` is at least this size.

```
int ReadEeprom(
    enum ADQEeprom  eeprom,
    uint32_t        address,
    void *const     buffer,
    size_t          length
)
```

Read data from the digitizer's EEPROM.

Return value

If the operation is successful, the return value is set to the number of bytes read from the digitizer's EEPROM. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function reads `length` bytes from the target EEPROM starting at the target `address` and writes the retrieved data to the `buffer`. Refer to Section 14 for more information.

Parameters

`eeprom (enum ADQEeprom)`

The target EEPROM area as a value from the enumeration `ADQEeprom`.

`address (uint32_t)`

The target address within the EEPROM area.

`buffer (void *const)`

A pointer to a memory region of (at least) size `length` to hold the data read from the EEPROM.

`length (size_t)`

The number of bytes to read from the EEPROM. It is implied that the memory region pointed to by `buffer` is at least this size.

A.4.10 Miscellaneous

`int SWTrig()` Thread-safe

Issue a software event.

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function issues an event from the software controlled event source. Refer to Section 6.2 for additional details.

`int Blink()`

Blink with the status LED.

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function blocks for five seconds while the status LED blinks blue with a 1 Hz on/off pattern.

```
int EjectTransferBuffer(
    int  channel
)
```
Thread-safe

Eject the transfer buffer(s).

Return value

If the operation is successful, `ADQ_EOK` is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function ejects the active transfer buffer for the specified channel making it available to the user immediately. The timing of the function call relative to data or external input is not guaranteed. This function is only intended to be used in specific cases, generally to eject the last buffer of an acquisition. Otherwise, use

- `eject_buffer_source` when the real-time relationship between the eject event and external inputs is important; or
- `eject_buffer_timeout` when transfer buffers should be ejected periodically.

Refer to Section 10.7 for more information.

Note

`EjectTransferBuffer()` will only eject partially filled buffers. Empty transfer buffers will not be ejected.

Parameters

`channel (int)`

The channel for which to eject any partially filled transfer buffer. The special value `ADQ_ANY_CHANNEL` is allowed and targets any channel with partially filled transfer buffers.

A.4.11 Development Kit

```
int ReadUserRegister(
    int        ul_target,
    uint32_t   regnum,
    uint32_t * retval
)
```

Thread-safe

Read from the register space of a user logic area.

Return value

If the operation is successful, ADQ_EOK is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function reads from the register space of one of the available user logic areas in the digitizer firmware. The function of a given register address is user defined through the *development kit*. For more information, refer to the development kit user guide. [8]

Parameters

`ul_target` (int)

The target user logic area. This value should be set to one of the alternatives in ADQUserLogic.

`regnum` (uint32_t)

The register address. Each increment corresponds to a 32-bit register index.

`retval` (uint32_t *)

A pointer to a memory region where the 32-bit read value will be stored.

```
int WriteUserRegister(
    int        ul_target,
    uint32_t   regnum,
    uint32_t   mask,
    uint32_t   data,
    uint32_t * retval
)
```

Thread-safe

Write to the register space of a user logic area.

Return value

If the operation is successful, ADQ_EOK is returned. A negative value indicates that an error has occurred. Refer to the trace log for more information about the cause of the error.

Description

This function writes to the register space of one of the available user logic areas in the digitizer firmware. The function of a given register address is user defined through the *development kit*. For more information, refer to the development kit user guide. [8]

Parameters

`ul_target (int)`

The target user logic area. This value should be set to one of the alternatives in `ADQUserLogic`.

`regnum (uint32_t)`

The register address. Each increment corresponds to a 32-bit register index.

`mask (uint32_t)`

A negative bit mask. Only the bits that are set to zero in this mask will be affected by the register write.

`data (uint32_t)`

The register write value.

`retval (uint32_t *)`

If a valid pointer to a memory region is provided via this parameter, the register will automatically be read after the write is completed, and the read data will be returned via this pointer. If this is not desired, a `NULL` pointer can be provided to prevent the readback.

A.5 Error Codes


```
#define ADQ_EOK (0)
```

Description

Signals the absence of any errors. The operation was successful.

```
#define ADQ_EINVAL (-1)
```

Description

The operation failed due to an invalid input value.

```
#define ADQ_EAGAIN (-2)
```

Description

The resource is temporarily unavailable. This is often used to indicate a timeout.

```
#define ADQ_EOVERFLOW (-3)
```

Description

The operation failed due to an overflow condition.

```
#define ADQ_ENOTREADY (-4)
```

Description

The resource is not yet ready.

```
#define ADQ_EINTERRUPTED (-5)
```

Description

The operation was interrupted.

```
#define ADQ_EIO (-6)
```

Description

The operation failed due to an input/output error.

```
#define ADQ_EEXTERNAL (-7)
```

Description

The operation failed due to an external error, e.g. from OS-level operations.

```
#define ADQ_EUNSUPPORTED (-8)
```

Description

The operation is unsupported.

```
#define ADQ_EINTERNAL (-9)
```

Description

The operation failed due to an internal error. This situation cannot be resolved by the user.

TELEDYNE SP DEVICES
Everywhereyoulook™

Worldwide Sales and Technical Support
spdevices.com

Teledyne SP Devices Corporate Headquarters
Teknikringen 8D
SE-583 30 Linköping
Sweden
Phone: +46 (0)13 645 0600
Fax: +46 (0)13 991 3044
Email: spd_info@teledyne.com