

ADQ3 Series FWDAQ

User Guide

Author(s): Teledyne SP Devices
Document ID: 21-2539
Classification: Public
Revision: C
Print date: 2021-06-21

Contents

1	Introduction	5
1.1	Overview	5
1.2	How to Read This Document	6
1.3	The First Acquisition	7
1.4	Definitions and Abbreviations	8
2	Analog Front-End	9
2.1	Input Range	9
2.2	Variable DC Offset	9
3	ADC	10
4	Clock System	11
4.1	Sampling Clock Generation	11
4.2	Clock Reference	11
5	Signal Processing	12
5.1	Digital Gain and Offset	12
5.2	Sample Skip	12
5.3	Digital Baseline Stabilization (DBS)	13
5.4	FIR Filter	14
5.4.1	Filter Design Example	14
6	Event Sources	17
6.1	Trigger Events	17
6.2	Software	17
6.3	Periodic	18
6.4	Signal Level	18
6.5	Port TRIG	19
6.6	Port SYNC	20
6.7	Port GPIOA	20
7	Functions	21
7.1	Pattern Generator	21
7.1.1	Operation	22
7.1.2	Count	22
7.1.3	Source	22
7.1.4	Reset Source	23
7.1.5	Output Value	23
7.1.6	Examples	23
7.2	Pulse Generator	26
7.3	Timestamp Synchronization	26

8	Ports	28
8.1	TRIG	28
8.1.1	Functions	29
8.2	SYNC	29
8.2.1	Functions	30
8.3	GPIOA	31
8.3.1	Functions	32
8.4	CLK	33
9	Data Acquisition	34
9.1	Timing Information	34
9.2	Starting and Stopping	36
9.3	Trigger Blocking	36
10	Data Transfer and Data Readout	37
10.1	Transfer Buffers	38
10.1.1	Advanced Parameters	38
10.2	Marker Buffers	39
10.2.1	Advanced Use Cases	39
10.3	Data Transfer	39
10.3.1	Interface	39
10.3.2	Program Flowchart	40
10.3.3	Record Data Transfer Buffer Format	43
10.3.4	Metadata Transfer Buffer Format	43
10.4	Data Readout	44
10.4.1	Interface	44
10.4.2	Record Buffers	44
10.4.3	Program Flowchart	44
10.5	Overflow	47
10.5.1	Physical Interface (case 1)	48
10.5.2	Transfer Interface (case 2)	48
10.6	Calculating the Data Rate	49
11	Test Pattern	50
12	System Manager	51
12.1	Firmware	51
12.1.1	Channel Configuration	51
12.2	Temperature Monitoring	52
13	Front Panel LEDs	53
13.1	STAT	53
13.2	RDY	53
13.3	USER	53

14 API	54
14.1 SDK Installation	55
14.1.1 Installing the SDK (Windows)	55
14.1.2 Installing the SDK (Linux)	55
14.2 Identification	56
14.3 Initialization	57
14.4 Configuration	58
14.5 Acquisition	58
14.6 Cleanup	60
14.7 Parameter Space	61
14.7.1 In Practice	62
14.7.2 JSON	63
15 Python API	65
15.1 Installation	66
15.2 Example	66
A API Reference	67
A.1 Defines	67
A.2 Enumerations	72
A.3 Structures	93
A.3.1 Initialization Parameters	95
A.3.2 Configuration Parameters	97
A.3.3 Status	137
A.3.4 Data	141
A.3.5 Other	145
A.4 Functions	146
A.4.1 General	147
A.4.2 Identification	148
A.4.3 Parameter Interface	151
A.4.4 Data Acquisition	160
A.4.5 Data Transfer	162
A.4.6 Data Readout	164
A.4.7 Status Monitoring	167
A.4.8 Cleanup	168
A.4.9 Miscellaneous	169
A.4.10 Development Kit	170
A.5 Error Codes	172

Document History

Revision	Date	Section	Description	Author
C	2021-06-21	4, 14	Add new clock system API and description, remove obsolete clock setup functions	TSPD
		5.4	Add section on FIR filter signal processing module	TSPD
		7.1	Document pattern generator reset behavior on <code>StartDataAcquisition()</code>	TSPD
		7.1.1	Use correct labels for the pattern generator instructions (was swapped)	TSPD
		9	Add note about the limitations of the trigger delay mechanism	TSPD
		12.1.1	Add section on channel configurations	TSPD
		14.2	Add missing argument in code snippet	TSPD
		14.7	Move parameter space description to its own section	TSPD
		14.7.2	Add section on JSON parameter API	TSPD
		A.3	Update reference manual to reflect changes to the parameter structs	TSPD
		A.4.10	Document development kit register access functions	TSPD
		1.3	Add Section 1.3	TSPD
		7.1.6	Fix parameter value in the trigger count example.	TSPD
B	2021-02-12	8.1, 8.2, 8.3	Add information about the input impedance.	TSPD
		-	Initial revision	TSPD
A	2021-02-08	-	Initial revision	TSPD

1 Introduction

This document is the user guide for ADQ3 series digitizers running the standard data acquisition firmware: FWDAQ.

Important

This document is only valid for the following digitizer models:

- ADQ32
- ADQ33

Note

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 digitizer's firmware, see Section 12 for more information. Refer to the datasheet for each digitizer model for information on which channel configurations are available.

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:

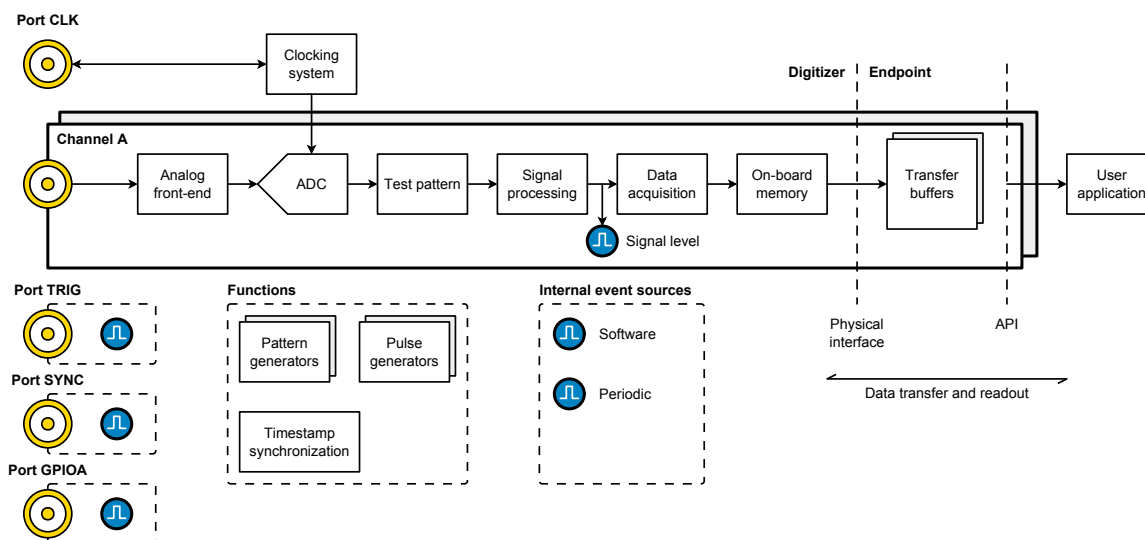



Figure 1: A block diagram presenting an overview of an ADQ3 series digitizer. The symbol  is used to indicate an event source (see Section 6).

- 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 application programming interface.
- Section [15](#) presents the Python wrapper for the application programming interface.
- Appendix [A](#) presents the API reference documentation.

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 [14](#)).

Text appearing in [this color](#) represent 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 [14.4](#), which outlines the configuration phase.

1.3 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 [14.1](#).
2. With the host computer powered off, mount the digitizer in an available PCIe slot 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 `adq3_series/data_readout` and follow the instructions in the README to compile. Do not make any changes to the example code for the first acquisition. This example follows the flow documented in Section [10.4.3](#).
4. Run the example application and verify that data is acquired.

1.4 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
DC	Direct current
DMA	Direct memory access
FFI	Foreign function interface
FIR	Finite impulse response
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.
Unbounded acquisition	A never-ending, or for practical purposes “infinite” acquisition. An acquisition can be unbounded 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].

Note

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 analog channels. For example, channel index 0 maps to channel A, index 1 maps to channel B.

2.1 Input Range

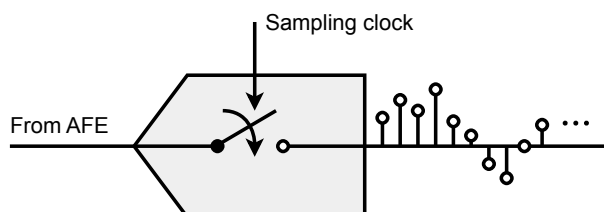
The input range determines which voltage range will map to the full scale range of the digitized samples, see Section 3, equation (1) for a conversion formula. The input range is fixed and cannot be modified on ADQ32 and ADQ33 digitizers. The current input range of the digitizer can be read programmatically 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). The lower bits should not be truncated in an attempt to match the ADC bit resolution. Signal processing modules will utilize the full 16-bit range and ignoring the least significant bits can result in a loss of accuracy.

Important

The 12-bit ADC data is MSB aligned to 16-bit values by the digitizer. Signal processing modules will utilize the full 16-bit range and ignoring the least significant bits in the output can result in a loss of accuracy.

The analog-to-digital conversion maps the range of the analog input of the digitizer onto the range of the 16-bit signed integers. To convert a digitized sample to a voltage, calculate

$$x_{volts} = \frac{x_{codes}}{2^{16}} \cdot \text{input_range} - \text{dc_offset} \quad (1)$$

where x_{codes} is a 16-bit sample output by the digitizer and x_{volts} is the corresponding value in volts. 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).

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

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

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 a 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 clock reference.

! Important

Reconfiguring the clock system parameters will reset parts of the data path and temporarily disrupt the clock, and should be considered part of device initialization. See Section 14.3.

4.1 Sampling Clock Generation

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

If full control over the sampling rate is needed, it is possible to disable the digitizer's internal clock generation and provide the full sampling clock via the CLK port directly, by setting the `clock_generator` to `ADQ_CLOCK_GENERATOR_EXTERNAL_CLOCK`.

4.2 Clock Reference

The clock reference 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 clock reference as `reference_source`. An alternative to using the internal clock reference is to let the digitizer lock onto an external clock reference provided via the CLK port, by setting the `reference_source` to `ADQ_CLOCK_REFERENCE_SOURCE_PORT_CLK`. It is also possible to output the digitizer's internal clock reference to other instruments via the CLK port, see Section 8.4.

When providing an external clock reference, the digitizer offers additional means to 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 clock reference path, prior to the sampling clock generator, which locks an internal 10 MHz oscillator to the clock reference, 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 clock reference path with a programmable `delay_adjustment` value, which allows precise tuning of the phase of the clock reference. This can be useful when building multi-digitizer systems with synchronization requirements. See the product datasheet for exact specifications on delay adjustment range [1] [2].

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)

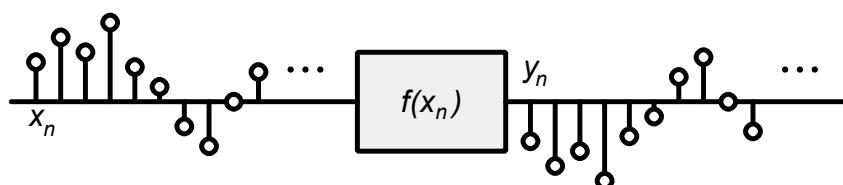


Figure 2: A typical signal processing module creates a new stream of data by applying some fixed operation to its input data stream.

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

$$y_n = \left(x_n \cdot \frac{\text{gain}}{\text{ADQ_UNITY_GAIN}} \right) + \text{offset} \quad (2)$$

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

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 block.

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 the `bypass` parameter set to 1.

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.

```
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)
taps = taps_fixed_point / 2 ** coefficient_fractional_bits
```

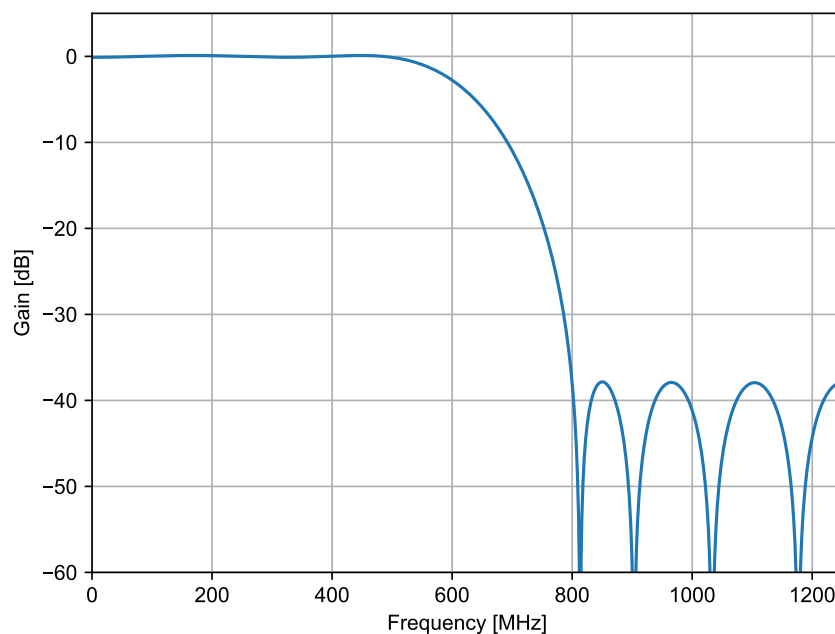


Figure 3: Example low-pass FIR filter frequency characteristic.

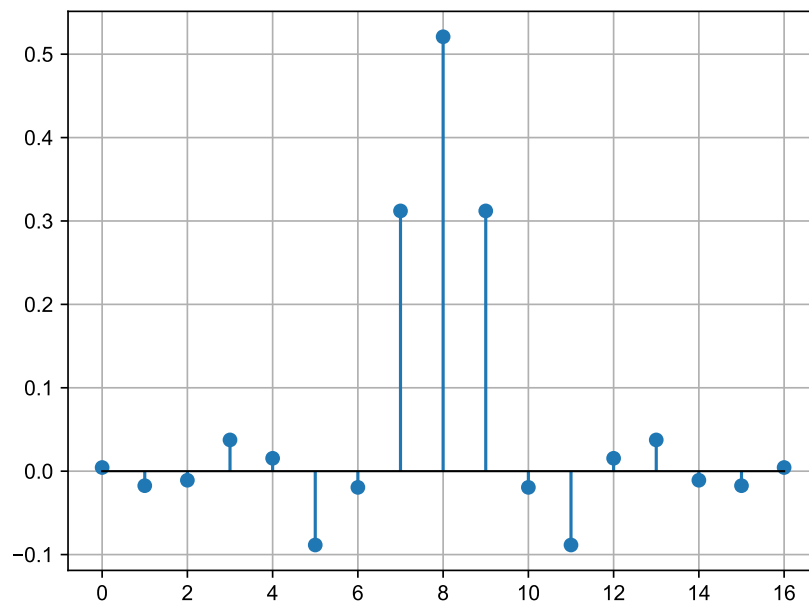


Figure 4: Example low-pass FIR filter impulse response.

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 source 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. 5 illustrates this relationship.

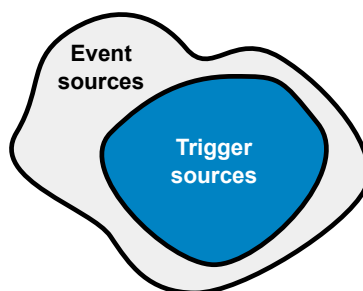


Figure 5: A trigger event always originates from an event source, but an event source may not always be used to generate trigger events.

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 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 close to 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`.

6.4 Signal Level

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

Important

Signal level events can only be used as trigger events.

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

$$\begin{aligned} \text{Arm level (rising)} &= \text{level} - \text{arm_hysteresis} \\ \text{Arm level (falling)} &= \text{level} + \text{arm_hysteresis} \end{aligned} \quad (3)$$

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. 6 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.

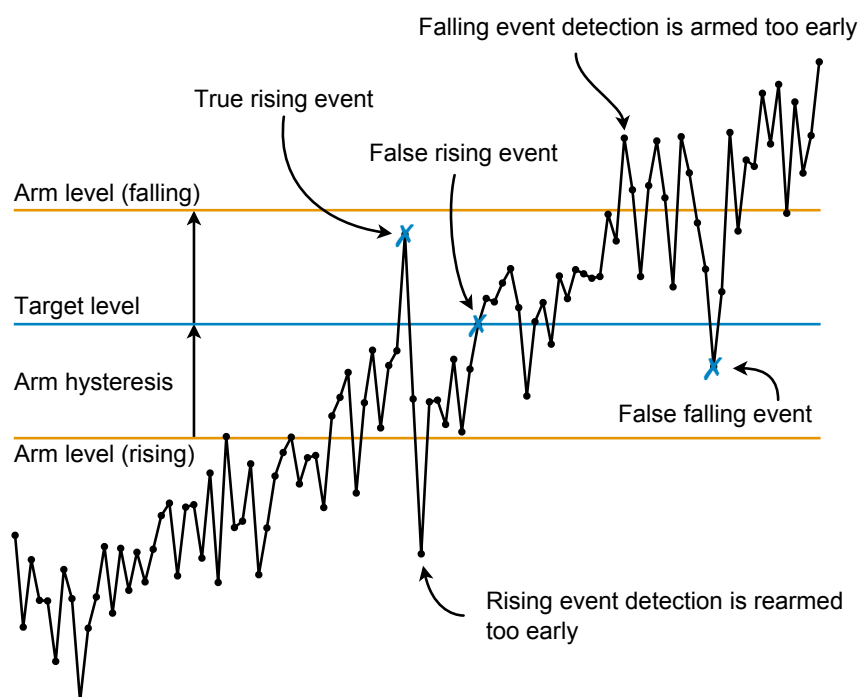


Figure 6: A demonstration of how a low arm hysteresis may cause false signal level events when analyzing a slow-moving noisy signal.

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 Port TRIG

The event source connected to the TRIG port performs edge detection of the connected signal with a configurable voltage `threshold`. The timing precision for these events is higher for the TRIG port compared to other ports such as SYNC or GPIOA. 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]. Refer to Section 8.1 and Fig. 10 for additional details on the TRIG port.

Example

For an ADQ32 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.

6.6 Port SYNC

The event source connected to the SYNC port performs edge detection of the connected 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]. Refer to Section 8.2 and Fig. 11 for additional details on the SYNC port.

Example

For an ADQ32 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.

6.7 Port GPIOA

The event source connected to the GPIOA port performs edge detection on the connected signal with a fixed voltage threshold. For exact details on the specifications of the GPIOA port when used as an event source, refer to the product datasheet [1] [2]. Refer to Section 8.2 and Fig. 11 for additional details on the GPIOA port.

Example

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

The GPIOA event source tied to the first (and only) pin is identified by the value `ADQ_EVENT_SOURCE_GPIOA0` and may be used as a trigger source.

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.

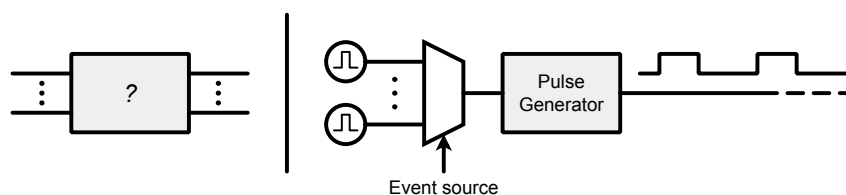


Figure 7: The definition of a function is general by design. They are opt-in and exist to meet specific use case requirements. One example is the pulse generator (Section 7.2) which is stimulated by an event source and outputs a digital pulse pattern. This signal can be output on any of the supported 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 9.3).

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 9.3), 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

$$\text{count_prescaling} \cdot \text{count}. \quad (4)$$

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. 8.

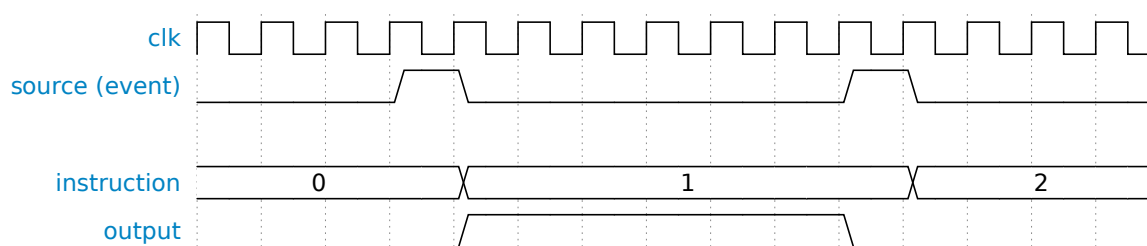


Figure 8: Timing diagram for two event instructions observing the same event source. Note the different transition values. Instruction 0 has `output_value = 0`, `output_value_transition = 0` and instruction 1 has `output_value = 1`, `output_value_transition = 0`.

7.1.6 Examples

This section provides a few examples of how to configure the pattern generator using the trigger blocking mechanism (Section 9.3) 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

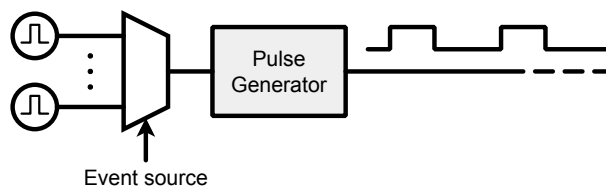
Configure the pattern generator to output logic high (block all triggers) until a rising edge event has been observed on the TRIG port. Following this, output logic low (accept all triggers) during a window of 100000 sampling periods. Any trigger event occurring as the window is opened will be accepted 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. 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`.

When an event from the selected event `source` matches the specified `edge` sensitivity, a pulse with a duration of `length` sampling periods is generated.

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`.

7.3 Timestamp Synchronization

The timestamp synchronization module is used to set the digitizer's internal timestamp counter (see Section 9.1) 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. 9.

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](#):

$$T_{n,disable} = T_0 + n \cdot T_p \quad \forall n \quad (5)$$

$$T_{n,first} = \begin{cases} T_0 & \text{if } n = 0 \\ \text{seed} + n \cdot T_p & \text{if } n > 0 \end{cases} \quad (6)$$

$$T_{n,all} = \begin{cases} T_0 & \text{if } n = 0 \\ \text{seed} + T_p & \text{if } n > 0 \end{cases} \quad (7)$$

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.

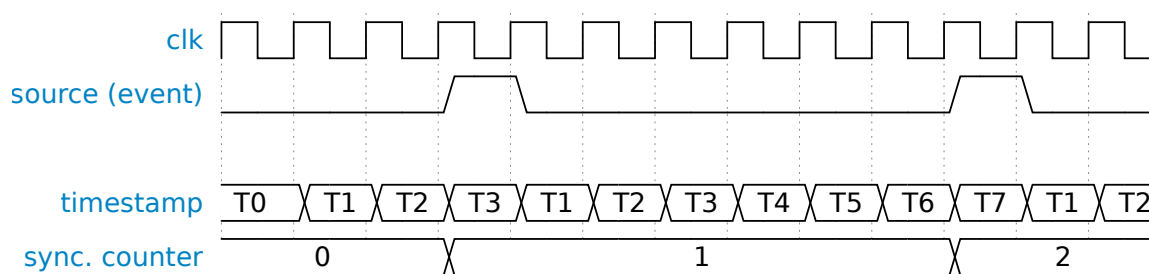


Figure 9: Timing diagram of the timestamp synchronization mechanism synchronizing on every event.

8 Ports

A *port* is a physical interface on the digitizer, excluding the analog inputs (the channels) and the device-to-host interface (the PCIe board connector). A port may consist of one or several *pins*. However, ADQ32 and ADQ33 only feature single pin ports. Unless otherwise specified, this is pin 0 when interfacing with the ports via the API (Section 14).

A port may offer a unique function set, a shared function 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 function, e.g. the CLK port exposes a signal path to the digitizer's clocking system (Section 4).

8.1 TRIG

The single pin TRIG port features high precision edge detection of an analog signal. The edge detection events may be used to trigger the acquisition of data (see Section 9) or to stimulate other parts of the digitizer. See Section 6.5 for information about the port's event source. Additionally, the port may be used to expose the output of *one* of the internal functions. See Section 8.1.1 for a list of supported output signals. A functional block diagram is shown in Fig. 10.

The port is either configured as an input or an output via the parameter *direction*, which effectively controls the state of the output buffer. When configured as an input, the user may select the *input_impedance*. When configured as an output, the output impedance is fixed. Refer to the product datasheet [1] [2] for the typical output impedance value.

The input path—and by extension the event source associated with the port—is always active. Thus, it is possible to output an internal signal while simultaneously performing edge detection and propagating the events to other parts of the digitizer. The current logic level is readable through the parameter *value*.

Note

By default, the port is configured as an input with an input impedance of 50 Ohms.

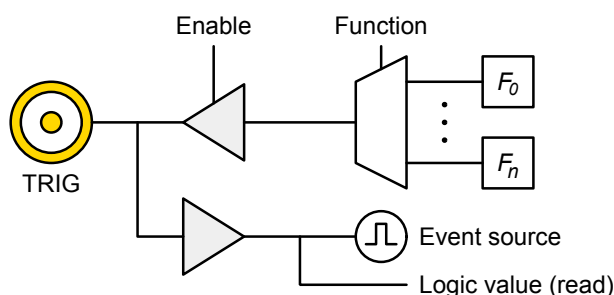


Figure 10: A functional block diagram of the TRIG port.

8.1.1 Functions

The TRIG port may expose the output of one of the internal functions (see Section 7). The possible output signals are fixed and are listed in Table 2.

Table 2: The output functions supported by the TRIG port.

Function	Section	API identifier
Disabled	-	ADQ_FUNCTION_INVALID
GPIO (port specific)	-	ADQ_FUNCTION_GPIO
Pattern generator 0	7.1	ADQ_FUNCTION_PATTERN_GENERATOR0
Pattern generator 1	7.1	ADQ_FUNCTION_PATTERN_GENERATOR1
Pulse generator 0	7.2	ADQ_FUNCTION_PULSE_GENERATOR0
Pulse generator 1	7.2	ADQ_FUNCTION_PULSE_GENERATOR1
Pulse generator 2	7.2	ADQ_FUNCTION_PULSE_GENERATOR2
Pulse generator 3	7.2	ADQ_FUNCTION_PULSE_GENERATOR3

Enable

- Set the `direction` to `ADQ_DIRECTION_OUT`.
- Set the `function` to an identifier from Table 2. If `ADQ_FUNCTION_GPIO` is selected, the output signal will transition to the logic level specified by the parameter `value`.
- Set the parameter `invert_output` to a nonzero value to invert the output signal. Set the parameter to zero to keep the source polarity.

Disable

- Set `direction` to `ADQ_DIRECTION_IN`.
- Set `function` to `ADQ_FUNCTION_INVALID`.

8.2 SYNC

Similar to the TRIG port (Section 8.1), the single pin SYNC port also features edge detection of an analog signal. However, the timing precision is not as high. The edge detection events may be used to trigger the acquisition of data (see Section 9) or to stimulate other parts of the digitizer. See Section 6.6 for information about the port's event source. Additionally, the port may be used to expose the output of *one* of the internal functions. See Section 8.2.1 for a list of supported output signals. A functional block diagram is shown in Fig. 11.

The port is either configured as an input or an output via the parameter `direction`, which effectively controls the state of the output buffer. When configured as an input, the user may select the

[input_impedance](#). When configured as an output, the output impedance is fixed. Refer to the product datasheet [1] [2] for the typical output impedance value.

The input path—and by extension the event source associated with the port—is always active. Thus, it is possible to output an internal signal while simultaneously performing edge detection and propagating the events to other parts of the digitizer. The current logic level is readable through the parameter [value](#).

Note

By default, the port is configured as an input with an input impedance of 50 Ohms.

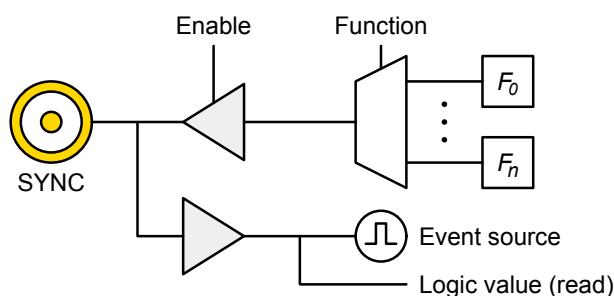


Figure 11: A functional block diagram of the SYNC port.

8.2.1 Functions

The SYNC port may expose the output of one of the internal functions (see Section 7). The possible output signals are fixed and are listed in Table 3.

Enable

- Set the [direction](#) to [ADQ_DIRECTION_OUT](#).
- Set the [function](#) to an identifier from Table 3. If [ADQ_FUNCTION_GPIO](#) is selected, the output signal will transition to the logic level specified by the parameter [value](#).
- Set the parameter [invert_output](#) to a nonzero value to invert the output signal. Set the parameter to zero to keep the source polarity.

Disable

- Set [direction](#) to [ADQ_DIRECTION_IN](#).
- Set [function](#) to [ADQ_FUNCTION_INVALID](#).

Table 3: The output functions supported by the SYNC port.

Function	Section	Identifier
Disabled	-	ADQ_FUNCTION_INVALID
GPIO (port specific)	-	ADQ_FUNCTION_GPIO
Pattern generator 0	7.1	ADQ_FUNCTION_PATTERN_GENERATOR0
Pattern generator 1	7.1	ADQ_FUNCTION_PATTERN_GENERATOR1
Pulse generator 0	7.2	ADQ_FUNCTION_PULSE_GENERATOR0
Pulse generator 1	7.2	ADQ_FUNCTION_PULSE_GENERATOR1
Pulse generator 2	7.2	ADQ_FUNCTION_PULSE_GENERATOR2
Pulse generator 3	7.2	ADQ_FUNCTION_PULSE_GENERATOR3

8.3 GPIOA

This section describes the general purpose input/output port A (GPIOA). The number of pins may differ between different digitizers in the ADQ3 family. Refer to the corresponding subsection for additional details.

ADQ32, ADQ33

The single pin GPIOA port features similar functionality to the SYNC port (Section 8.2). Edge detection of an analog signal is supported and the events may be used to trigger the acquisition of data (see Section 9) or to stimulate other parts of the digitizer. See Section 6.6 for information about the port's event source. Additionally, the port may be used to expose the output of *one* of the internal functions. See Section 8.3.1 for a list of supported output signals. A functional block diagram is shown in Fig. 12.

The port is either configured as an input or an output via the parameter `direction`, which effectively controls the state of the output buffer. When configured as an input, the user may select the `input_impedance`. When configured as an output, the output impedance is fixed. Refer to the product datasheet [1] [2] for the typical output impedance value.

The input path—and by extension the event source associated with the port—is always active. Thus, it is possible to output an internal signal while simultaneously performing edge detection and propagating the events to other parts of the digitizer. The current logic level is readable through the parameter `value`.

Note

By default, the port is configured as an input with an high input impedance. Refer to the product datasheet [1] [2] for the typical value.

Note

This port is labeled “GPIO” on the front panel.

8.3.1 Functions

The GPIOA port may expose the output of one of the internal functions (see Section 7). The possible output signals are fixed and are listed in Table 4.

Enable

- Set the `direction` to `ADQ_DIRECTION_OUT`.
- Set the `function` to an identifier from Table 4. If `ADQ_FUNCTION_GPIO` is selected, the output signal will transition to the logic level specified by the parameter `value`.
- Set the parameter `invert_output` to a nonzero value to invert the output signal. Set the parameter to zero to keep the source polarity.

Disable

- Set `direction` to `ADQ_DIRECTION_IN`.
- Set `function` to `ADQ_FUNCTION_INVALID`.

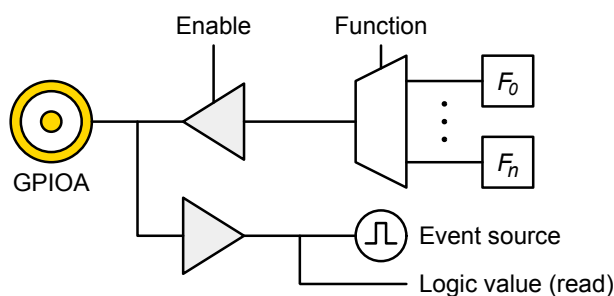


Figure 12: A functional block diagram of the GPIOA port on ADQ32 and ADQ33. The port is labeled “GPIO” on the front panel.

Table 4: The output functions supported by the GPIOA port.

Function	Section	Identifier
Disabled	-	<code>ADQ_FUNCTION_INVALID</code>
GPIO (port specific)	-	<code>ADQ_FUNCTION_GPIO</code>
Pattern generator 0	7.1	<code>ADQ_FUNCTION_PATTERN_GENERATOR0</code>
Pattern generator 1	7.1	<code>ADQ_FUNCTION_PATTERN_GENERATOR1</code>
Pulse generator 0	7.2	<code>ADQ_FUNCTION_PULSE_GENERATOR0</code>
Pulse generator 1	7.2	<code>ADQ_FUNCTION_PULSE_GENERATOR1</code>
Pulse generator 2	7.2	<code>ADQ_FUNCTION_PULSE_GENERATOR2</code>
Pulse generator 3	7.2	<code>ADQ_FUNCTION_PULSE_GENERATOR3</code>

8.4 CLK

The single pin CLK port exposes a signal path to the digitizer's clocking system (see Section 4). By default, the port is configured as an input with an input impedance of 50 Ohms. The parameter `input_impedance` may be set to `ADQ_IMPEDANCE_HIGH` to enable a high impedance mode. Refer to the product datasheet [1] [2] for typical values. Setting the parameter `direction` to `ADQ_DIRECTION_OUT` configures the port as an output, and causes the clock system to start driving the digitizer's internal 10 MHz reference signal. A functional block diagram is shown in Fig. 13.

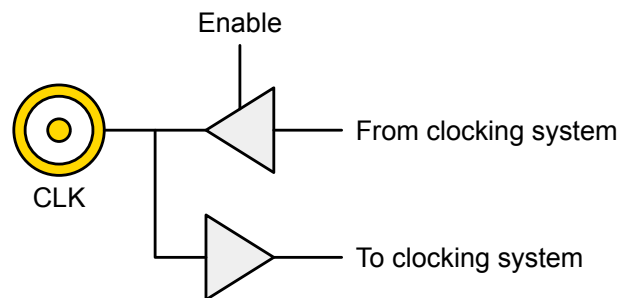


Figure 13: A functional block diagram of the CLK port.

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.

The acquisition parameters consists of

- the number of records to acquire,
- the record length,
- the trigger event source and its edge (see Section 6),
- the horizontal offset; and
- the trigger blocking source (see Section 9.3).

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 unbounded acquisition. For each active channel, the `record_length` is expected to be a nonzero value. Only records with a fixed length are supported. 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. 14 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, i.e. 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.

9.1 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`. 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].

Important

The timing resolution varies between event sources.

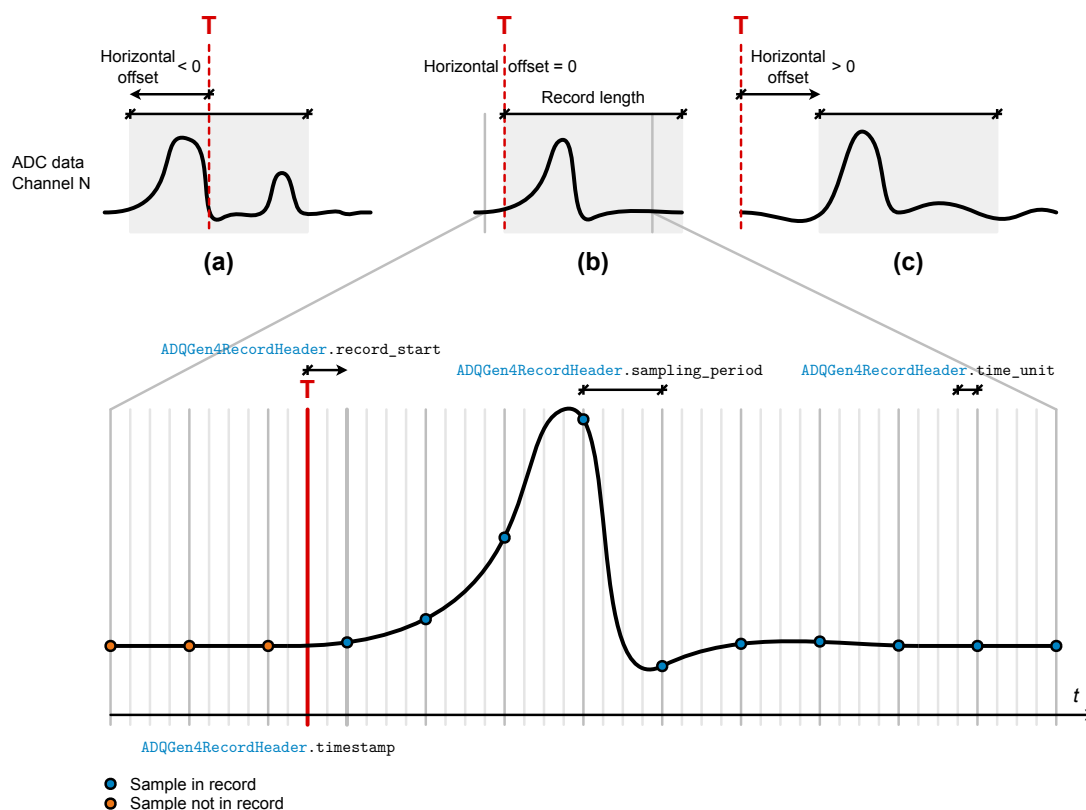


Figure 14: Illustration of how the parameters of the data acquisition process affect the acquisition of a record (the shaded region). Relative values, e.g. the record header field `record_start`, are all anchored in the trigger event **T**. The header field `timestamp` specifies where the trigger event is located on the timing grid.

9.2 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.

9.3 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. Trigger events occurring while the output of this function is logic high will be ignored by the data acquisition process. The 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 a trigger blocking source.

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 (within the API) 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 10.4. Fig. 15 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 10.4.

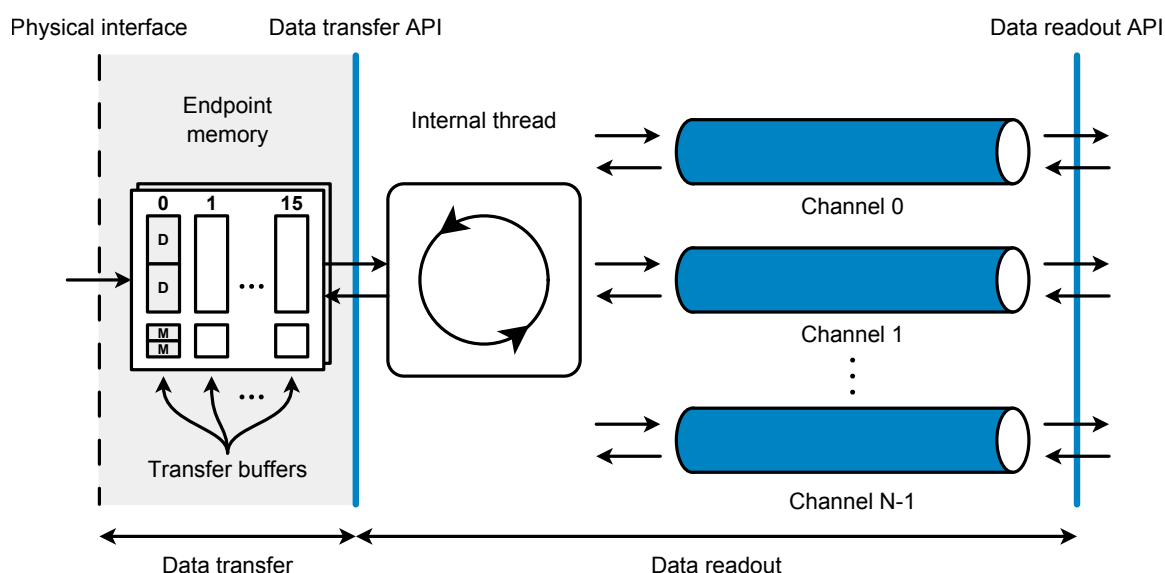


Figure 15: Overview of the data transfer and data readout processes. Data is transferred over the physical interface to transfer buffers in the endpoint memory. If the transfer buffers reside in the host computer's RAM, the data readout process is available as an optional layer. In this case, an internal 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 meta-data 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.

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.

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 [10.4](#)), 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, i.e. buffer 1 will be written to 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 10.4) 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 10.4).

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 `adq3_series/data_transfer_gpu_amd`. See Section 14.1 for details on how to locate this software example.

10.3 Data Transfer

Important

It is recommended to begin developing the user application by using the data readout interface described in Section 10.4 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.

10.3.1 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.

10.3.2 Program Flowchart

The expected program logic for a user application reading data directly via the data transfer interface is presented in Fig. 16. 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

A source code example demonstrating the program flow for the configuration in step 1a is available for Windows platforms as `adq3_series/data_transfer_gpu_nvidia_through_host`. See Section [14.1](#) for details on how to locate this software example.

Note

A source code example demonstrating the program flow for the configuration in step 1b is available for Linux platforms as `adq3_series/data_transfer_gpu_nvidia`. See Section [14.1](#) 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 [14.4](#). 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 `record_length_infinite_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).

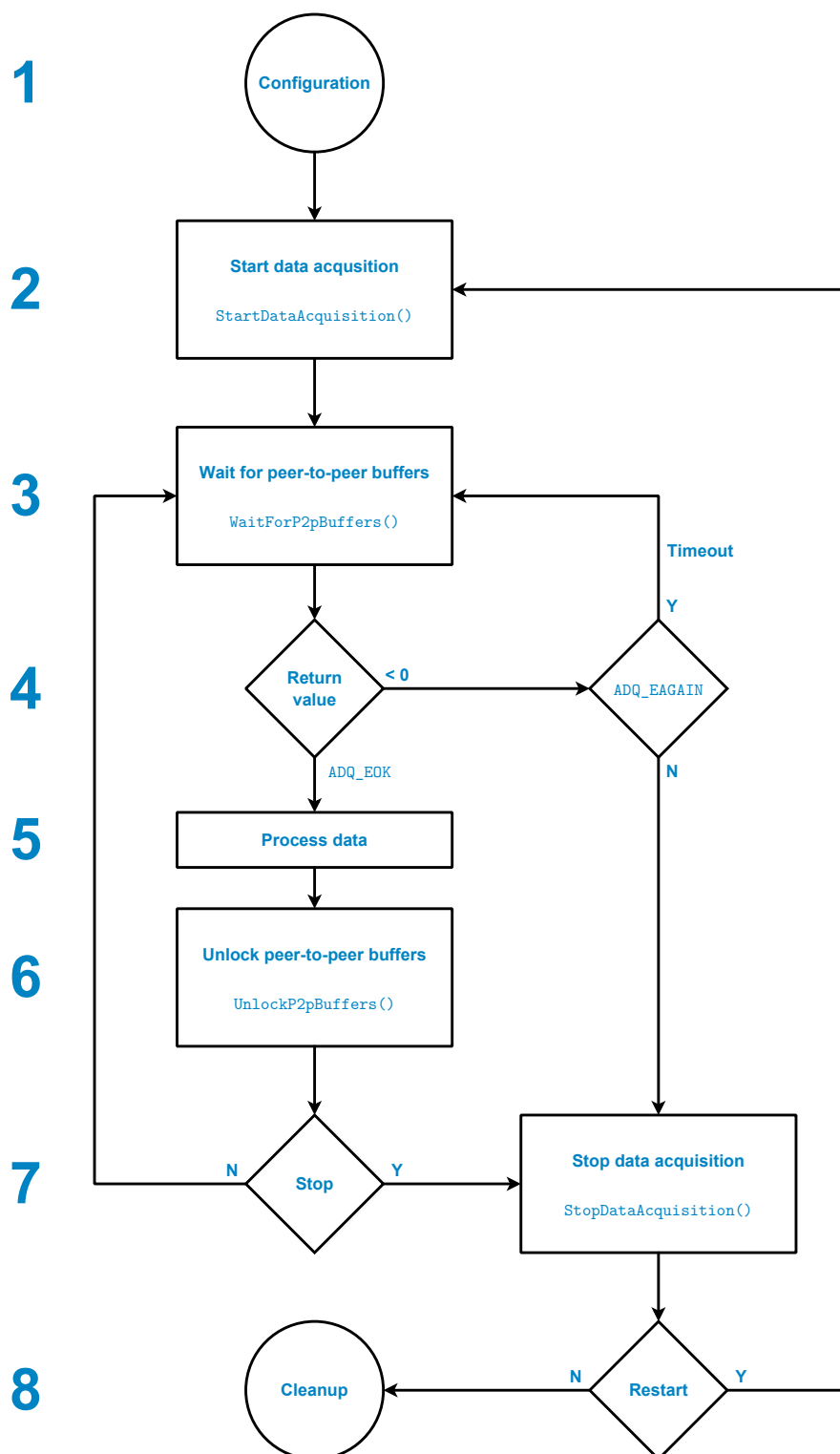


Figure 16: A flowchart for a user application interfacing directly with the data transfer process. The steps are labeled on the left-hand side and have a matching entry in Section 10.3.2.

(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 14.4. 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`].
 - 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 `record_length_infinite_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 10.5). 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 10.5 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 14.6.

10.3.3 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.

10.3.4 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 disabled 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.

10.4 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.

! Important

The data readout interface is only supported when the transfer buffers are located in the host computer's RAM.

10.4.1 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.

10.4.2 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

A record buffer always contains data from a full record. `WaitForRecordBuffer()` does not yet support returning partial record data.

10.4.3 Program Flowchart

The expected program flow for a user application reading data via the data readout interface is presented in Fig. [17](#). The steps are labeled on the left-hand side and are explained in the following list.

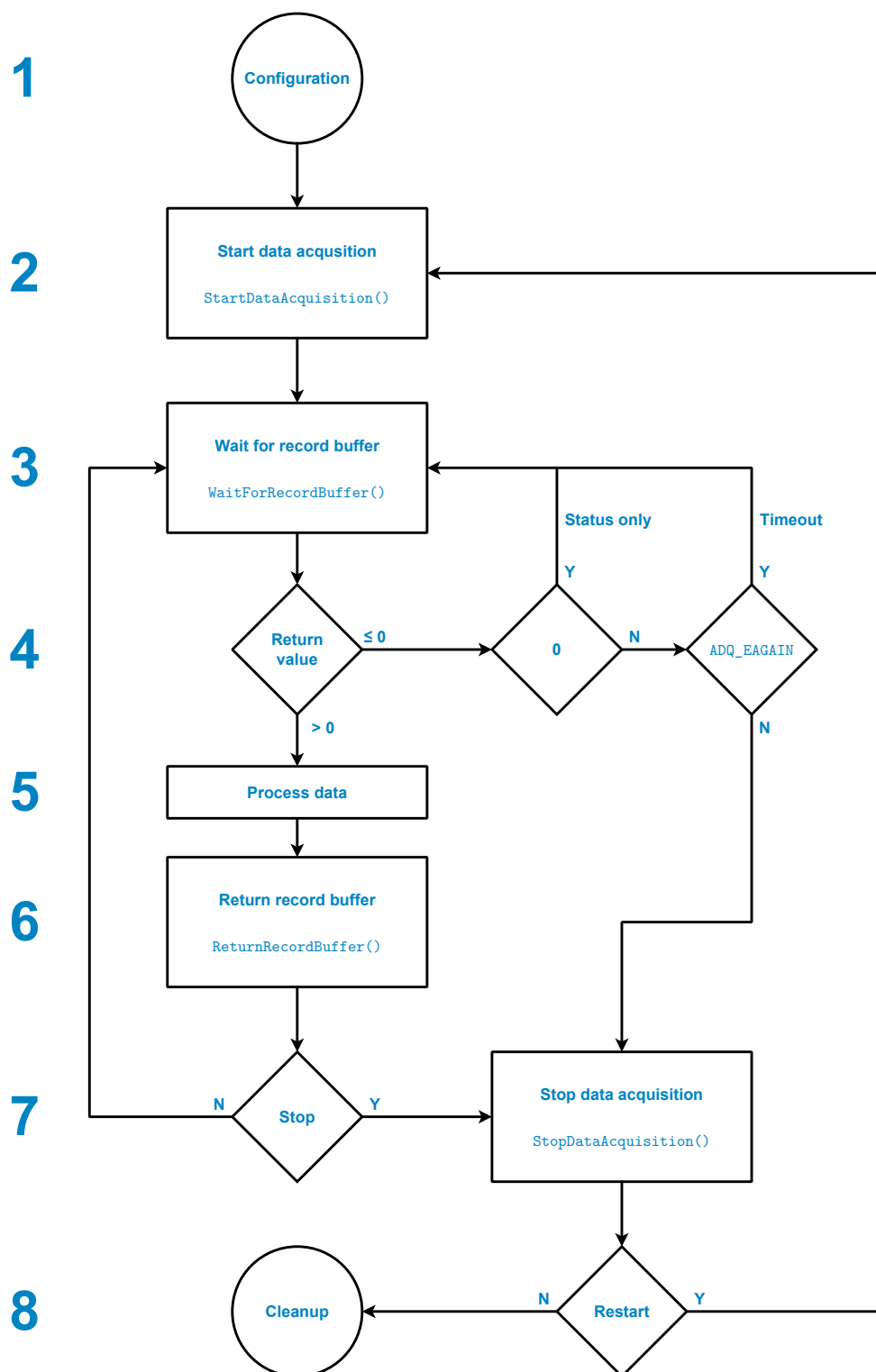


Figure 17: A flowchart for the data readout process. The steps are labeled on the left-hand side and have a matching entry in Section 10.4.3.

Note

The program flow in Fig. 17 is implemented in the software example `adq3_series/data_readout`. This example code in C is available as part of the software development kit (SDK). See Section 14.1 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 14.4. 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`].
 - Decide the number of records per transfer buffer (positive integer). Let this value be N.
 - Set `record_size` to the expected size (in bytes) of a record.
 - Set `record_length_infinite_enabled` to 0 (default).
 - Set `record_buffer_size` to N times the record size.
 - Set `metadata_enabled` to 1.
 - Set `metadata_buffer_size` to N times the size of a record header `ADQGen4RecordHeader`.
- 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. 15) 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()`.

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 10.5).
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 10.5 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 14.6.

10.5 Overflow

❗ Important

The data acquisition stops in the event of an overflow.

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 10.5.1 and 10.5.2 describes the two cases in more detail. Refer to `GetStatus()` for information on how to query the digitizer for the overflow status.

10.5.1 Physical Interface (case 1)

The 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 come to a halt. The contents of the on-board memory remain intact and can be transferred safely, but no new records will be created. Refer to [GetStatus\(\)](#) for information on how to query the digitizer for the overflow status.

Note

The overflow behavior will change with later releases of the ADQ32 and ADQ33 firmware.

10.5.2 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 record buffer propagated via [WaitForRecordBuffer\(\)](#), a corresponding call to [ReturnRecordBuffer\(\)](#) returning the memory is expected. The same applies to [WaitForP2pBuffers\(\)](#) and [UnlockP2pBuffers\(\)](#).

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 10.5.1. 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 10.3.2, step 5 and Section 10.4.3, 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 10.3.2, step 5 and Section 10.4.3, step 5) is able to handle the acquisition data rate.

10.6 Calculating the Data Rate

Equation (8) presents how to calculate the effective data rate of *one channel*, assuming two bytes per sample, 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.

$$2 \cdot \text{record_length} \cdot f_{trigger} \quad [\text{bytes/s}] \quad (8)$$

If metadata is enabled (`metadata_enabled` is a nonzero value), each trigger also adds the transfer of an `ADQGen4RecordHeader`. Thus, (8) becomes

$$(2 \cdot \text{record_length} + \text{sizeof}(\text{ADQGen4RecordHeader})) \cdot f_{trigger} \quad [\text{bytes/s}] \quad (9)$$

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 (9), 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 (8), 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

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

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 be 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 and supply voltage monitoring, overtemperature protection and fan control.

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 [3] for more information.

If the digitizer firmware has somehow been compromised and communication via PCIe is not possible, a fallback option is provided via the USB-C connector located at the top edge of the digitizer. The USB-C connector connects directly to the system manager and allows firmware programming regardless of the current state of the digitizer firmware.

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 (13376024 B)
      Revision: 1.1-RC1-835bce4
      Description: 2CH-FWDAQ
      Part number: 400-023-001
      MD5: C05953A35949D9824A57162511C4A006
      Address: 0x02000000
      Timestamp: 2021-06-16T12:05:33Z

Image 1
-----
      Size: 13 MB (13442856 B)
      Revision: 1.1-RC1-835bce4
      Description: 1CH-FWDAQ
      Part number: 400-023-000
      MD5: D3081A799E08B1C54F9B027954B627D7
      Address: 0x01000000
      Timestamp: 2021-06-16T12:09:21Z
```

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 Temperature Monitoring

The system manager continually monitors the temperatures of components in the digitizer hardware such as the FPGA, ADCs and supply voltage converters. Each temperature sensor has a set limit, which once exceeded, will cause the system manager to go into an overtemperature protection state. In this state, supply voltages are disabled to allow the digitizer to cool down to avoid damaging system components. Once in the overtemperature state, the digitizer will not be able to operate properly until it has been completely power cycled. The overtemperature condition can be detected via the STAT LED, see Section 13.1. The current values of the temperature sensors can be accessed through the [sensor](#) entries in [ADQTemperatureStatus](#), see [GetStatus\(\)](#).

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 5 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 5: The states of the STAT LED, sorted in descending priority.

LED state	Description
Blinking red, 5 Hz	A temperature sensor reached its limit and has triggered the overtemperature protection. Supplies will have been disabled to avoid damage to the device, and a power cycling of the system is required to exit the state.
Blinking red, alternating long/short	Waiting for power supplies to start up.
Constant purple	Failed to communicate with the digitizer's firmware memory.
Blinking purple, 1 Hz	One of the temperature or voltage sensors is unreadable.
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.

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 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 [14.2](#))
2. Initialization (Section [14.3](#))
3. Configuration (Section [14.4](#))
4. Acquisition (Section [14.5](#))
5. Cleanup (Section [14.6](#))

The flow between these parts are visualized in Fig. [18](#) 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 [14.7](#) for detailed information on interacting with the parameter space.

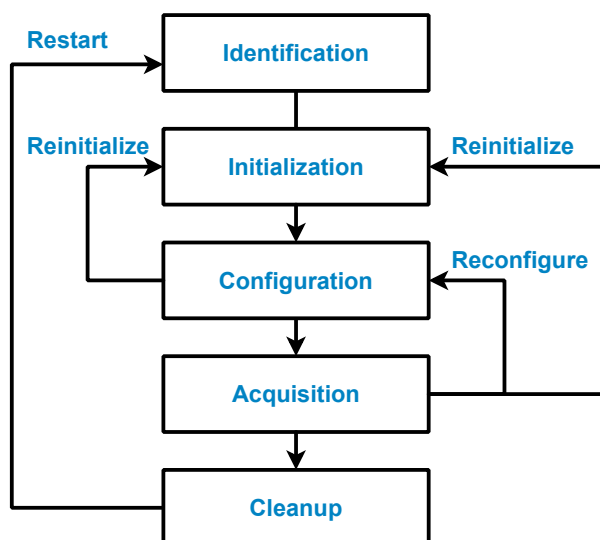


Figure 18: Typical control flow of an application interfacing with an ADQ3 series digitizer.

14.1 SDK Installation

The Software Development Kit (SDK) contains the ADQAPI, drivers, examples and documentation required to successfully interface with the digitizer. The installation procedure for Microsoft Windows and Linux is described in the following sections.

14.1.1 Installing the SDK (Windows)

For Microsoft Windows the SDK is installed by running

```
TSPD-SDK-installer_rXXXXX.exe
```

and following the instructions. The XXXXX part of the file name is the version number. After the installation the example code is located in

```
<Path to installation directory>/C_examples/
```

and the documentation in

```
<Path to installation directory>/Documentation/
```

14.1.2 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 [4]. The installation files are included in

```
ADQ_SDK_linux_rXXXXX.tar.gz
```

where XXXXX is the version number. The archive contains installation files, example code and documen-

tation. The README file, located in the root directory of the archive, describes the installation procedure in detail for the different distributions.

14.2 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 15 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)
    != ADQ_EOK)
    /* Handle error */
```

Note that the identification number (`adq_num`) used in device calls starts at 1.

14.3 Initialization

In the initialization phase, parameter changes that disrupt the digitizer's operation are performed. See Section 14.7 for details on interacting with the parameter space.

Currently, the only example of an initialization phase parameter is changing the clock system configuration via [ADQClockSystemParameters](#). Changing the clock system setup 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 clock reference and clock generator, which means that this step can be safely skipped if this is the desired clock system configuration.

Note

The initialization step may be safely skipped if the digitizer should use the internal reference and the default base sampling rate.

The code snippet below sets up the clock system for an external clock reference 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_CLOCK_REFERENCE_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 clock reference 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_CLOCK_REFERENCE_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 clock reference:

```
/* Use external clock reference with delay adjustment
   enabled and 100ps added delay. */
clock_system.reference_source = ADQ_CLOCK_REFERENCE_SOURCE_PORT_CLK;
clock_system.reference_frequency = 10e6;
clock_system.delay_adjustment_enabled = 1;
clock_system.delay_adjustment = 100e-12;
```

14.4 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 14.5). 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 14.3). See Section 14.7 for details on interacting with the parameter space.

14.5 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 14.4) and the data transfer parameters have been configured according to Section 10.4. 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 10.4.3, step 5 for more information about important things 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;
}
```

14.6 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);
```

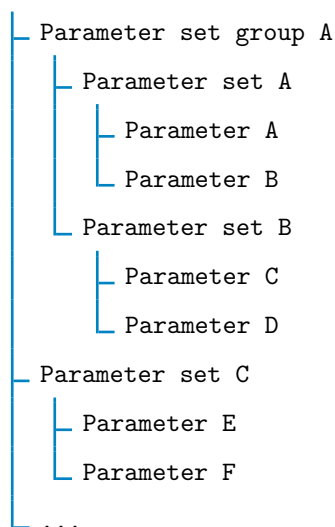
which will delete the control unit and all device objects. The memory allocated by the API, e.g. the record buffers, will also be freed. Any attempt to read this memory after the control unit has been deleted will cause access violations. To reconnect to the digitizer, the steps outlined in the identification phase (Section 14.2) must be executed.

14.7 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 digitizers parameters involves this tree structure and four functions:

- `InitializeParameters()`
- `GetParameters()`
- `ValidateParameters()`
- `SetParameters()`

From the perspective of the user application, these functions transports 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.

14.7.1 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 struct definitions that together build up the nodes of the parameter tree described in Section 14.7.

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 */
}
```

14.7.2 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_CLOCK_REFERENCE_SOURCE_INTERNAL",
  "sampling_frequency": 5000000000,
  "reference_frequency": 10000000,
  "delay_adjustment": 0,
  "low_jitter_mode_enabled": 1,
  "delay_adjustment_enabled": 0
}
```

15 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 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 [14.7.2](#) 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`

15.1 Installation

The pyadq package can be installed to the directory for user-installed packages by executing

```
$ pip install --user <path to pyadq>
```

15.2 Example

An example for the ADQ3 series digitizers is included and demonstrates how to set up and acquire data from a single digitizer. The example can be found at the following location:

```
example/adq3_series.py
```

References

- [1] Teledyne Signal Processing Devices Sweden AB, *20-2378 ADQ32 datasheet*. Technical Specification.
- [2] Teledyne Signal Processing Devices Sweden AB, *20-2451 ADQ33 datasheet*. Technical Specification.
- [3] Teledyne Signal Processing Devices Sweden AB, *18-2059 ADQUpdater User Guide*. Technical Manual.
- [4] Teledyne Signal Processing Devices Sweden AB, *15-1494 Digitizer OS Support*. Technical Specification.
- [5] Teledyne Signal Processing Devices Sweden AB, *14-1351 ADQAPI Reference Guide*. Technical Manual.
- [6] 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.

<code>ADQAPI_VERSION_MAJOR</code>	68
<code>ADQAPI_VERSION_MINOR</code>	68
<code>ADQ_MAX_NOF_CHANNELS</code>	68
<code>ADQ_MAX_NOF_BUFFERS</code>	68
<code>ADQ_MAX_NOF_PORTS</code>	68
<code>ADQ_MAX_NOF_PINS</code>	68
<code>ADQ_MAX_NOF_ADC_CORES</code>	68
<code>ADQ_MAX_NOF_INPUT_RANGES</code>	69
<code>ADQ_MAX_NOF_PATTERN_GENERATORS</code>	69
<code>ADQ_MAX_NOF_PULSE_GENERATORS</code>	69
<code>ADQ_MAX_NOF_PATTERN_INSTRUCTIONS</code>	69
<code>ADQ_MAX_NOF_TEMPERATURE_SENSORS</code>	69
<code>ADQ_MAX_NOF_FILTER_COEFFICIENTS</code>	69
<code>ADQ_ANY_CHANNEL</code>	69
<code>ADQ_INFINITE_RECORD_LENGTH</code>	70
<code>ADQ_INFINITE_NOF_RECORDS</code>	70
<code>ADQ_UNITY_GAIN</code>	70
<code>ADQ_PARAMETERS_MAGIC</code>	70
<code>LOG_LEVEL_ERROR</code>	70
<code>LOG_LEVEL_WARN</code>	70
<code>LOG_LEVEL_INFO</code>	71

```
#define ADQAPI_VERSION_MAJOR 4
```

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_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

Reserved

```
#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_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 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()`.

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.

ADQParameterId	73
ADQStatusId	77
ADQEventSource	78
ADQTestPatternSource	81
ADQPort	82
ADQImpedance	84
ADQDirection	84
ADQEdge	84
ADQClockGenerator	85
ADQClockReferenceSource	85
ADQFunction	86
ADQPatternGeneratorOperation	87
ADQMarkerMode	87
ADQMemoryOwner	88
ADQTimestampSynchronizationMode	88
ADQTimestampSynchronizationArm	89
ADQCommunicationInterface	89
ADQCoefficientFormat	90
ADQRoundingMethod	90
ADQUserLogic	91
ADQHWIFEnum	91
ADQProductID_Enum	92

```

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_SYNCO              = 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_CLOCK_SYSTEM            = 40
}
  
```

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 [ADQDataAcquisition-Parameters](#).

ADQ_PARAMETER_ID_DATA_TRANSFER (2)

The identification number for the data transfer parameters, defined by [ADQDataTransfer-Parameters](#).

ADQ_PARAMETER_ID_DATA_READOUT (3)

The identification number for the data readout parameters, defined by [ADQDataReadout-Parameters](#).

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 digital gain and offset parameters, defined by [ADQDigitalGain-AndOffsetParameters](#).

ADQ_PARAMETER_ID_EVENT_SOURCE_LEVEL (6)

The identification number for the level event source parameters, defined by [ADQEventSource-LevelParameters](#).

ADQ_PARAMETER_ID_DBS (7)

The identification number for the digital baseline stabilization parameters, defined by [ADQDbs-Parameters](#).

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 [ADQAnalogFrontendParameters](#).

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 [ADQEventSourceParameters](#).

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 [ADQFunctionParameters](#).

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_SYNC0 (22)

The identification number for the parameters associated with the SYNC0 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 [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_CLKI (25)

The identification number for the parameters associated with the CLKI port, defined by [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_CLKO (26)

The identification number for the parameters associated with the CLKO port, defined by [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_GPIOA (27)

The identification number for the parameters associated with the GPIOA port, defined by [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_GPIOB (28)

The identification number for the parameters associated with the GPIOB port, defined by [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_PXIE (29)

The identification number for the parameters associated with the PXIE port, defined by [ADQPort-Parameters](#).

ADQ_PARAMETER_ID_PORT_MTCA (30)

The identification number for the parameters associated with the MTCA port, defined by [ADQPort-Parameters](#).

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 [ADQPulse-GeneratorParameters](#).

ADQ_PARAMETER_ID_PULSE_GENERATOR2 (33)

The identification number for the parameters of the third pulse generator, defined by [ADQPulse-GeneratorParameters](#).

ADQ_PARAMETER_ID_PULSE_GENERATOR3 (34)

The identification number for the parameters of the fourth pulse generator, defined by [ADQPulse-GeneratorParameters](#).

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_CLOCK_SYSTEM (40)

The identification number for the parameters of the clock system defined by [ADQClockSystemParameters.](#)

```
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
}
```

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 [ADQTemperatureStatus.](#)

```
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_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
}
```

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 [5] 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)

Reserved

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_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.

```
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
}
```

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)

A constant specifying the GPIOB port. Unused on ADQ3 series digitizers.

ADQ_PORT_PXIE (9)

A constant specifying the PXIE port. Unused on ADQ3 series digitizers.

ADQ_PORT_MTCA (10)

A constant specifying the MTCA port. Unused on ADQ3 series digitizers.

```
enum ADQImpedance {
    ADQ_IMPEDANCE_50_OHM = 0,
    ADQ_IMPEDANCE_HIGH   = 1
}
```

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 [1] [2] for typical values.

```
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 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 ADQClockReferenceSource {  
    ADQ_CLOCK_REFERENCE_SOURCE_INTERNAL    = 1,  
    ADQ_CLOCK_REFERENCE_SOURCE_PORT_CLK    = 2,  
    ADQ_CLOCK_REFERENCE_SOURCE_PXIE_10M    = 3,  
    ADQ_CLOCK_REFERENCE_SOURCE_MTCA_TCLKA  = 4,  
    ADQ_CLOCK_REFERENCE_SOURCE_MTCA_TCLKB  = 5,  
    ADQ_CLOCK_REFERENCE_SOURCE_PXIE_100M   = 6  
}
```

Description

An enumeration of the digitizer's clock reference sources. Not all digitizers support every clock reference source. See Section 4.2 for details.

Values

`ADQ_CLOCK_REFERENCE_SOURCE_INTERNAL` (1)

A constant specifying the internal clock reference.

ADQ_CLOCK_REFERENCE_SOURCE_PORT_CLK (2)

A constant specifying an external clock reference supplied via the CLK port.

ADQ_CLOCK_REFERENCE_SOURCE_PXIE_10M (3)

A constant specifying the 10 MHz clock reference in a PXIe backplane. Only available for PXIe form factor digitizers.

ADQ_CLOCK_REFERENCE_SOURCE_MTCA_TCLKA (4)

A constant specifying the TCLKA clock reference in a MicroTCA backplane. Only available for MTCA form factor digitizers.

ADQ_CLOCK_REFERENCE_SOURCE_MTCA_TCLKB (5)

A constant specifying the TCLKB clock reference in a MicroTCA backplane. Only available for MTCA form factor digitizers.

ADQ_CLOCK_REFERENCE_SOURCE_PXIE_100M (6)

A constant specifying the 100 MHz clock reference 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
}
```

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.

```
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 receiving node 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 ADQTimestampSynchronizationArm {  
    ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_IMMEDIATE    = 0,  
    ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_ACQUISITION  = 1  
}
```

Description

An enumeration specifying when the timestamp synchronization should be armed.

Values

ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_IMMEDIATE (0)

A constant specifying that the timestamp synchronization should be armed in the call to [SetParameters\(\)](#).

ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_ACQUISITION (1)

A constant specifying that the timestamp synchronization should be armed in the call to [StartDataAcquisition\(\)](#).

```
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 ADQHWIFEnum {  
    HWIF_USB      = 0,  
    HWIF_PCIE     = 1,  
    HWIF_USB3     = 2,  
    HWIF_PCIE-lite = 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_ADQ3        = 0x0031,
    PID_ADQSM       = 0x0032,
    PID_TX320       = 0x201A,
    PID_RX320       = 0x201C,
    PID_S6000       = 0x2019
}
```

Description

An enumeration of the product IDs.

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_MINOR`. See `ADQAPI_ValidateVersion()` for more information about how to implement version control the user application space.

Initialization Parameters	95
ADQClockSystemParameters	95
Configuration Parameters	97
ADQParameters	97
ADQAnalogFrontendParameters	98
ADQAnalogFrontendParametersChannel	99
ADQConstantParameters	100
ADQConstantParametersChannel	102
ADQConstantParametersPort	103
ADQConstantParametersPin	103
ADQConstantParametersCommunicationInterface	104
ADQConstantParametersFirFilter	105
ADQDataAcquisitionParameters	105
ADQDataAcquisitionParametersCommon	106
ADQDataAcquisitionParametersChannel	107
ADQDataTransferParameters	108
ADQDataTransferParametersCommon	109
ADQDataTransferParametersChannel	111
ADQDataReadoutParameters	113
ADQDataReadoutParametersCommon	114
ADQDataReadoutParametersChannel	114
ADQDbParameters	115
ADQDbParametersChannel	116
ADQDigitalGainAndOffsetParameters	116
ADQDigitalGainAndOffsetParametersChannel	117
ADQFirFilterParameters	118
ADQFirFilterParametersChannel	118
ADQEventSourceParameters	119
ADQEventSourceLevelParameters	120
ADQEventSourceLevelParametersChannel	121
ADQEventSourcePeriodicParameters	121
ADQEventSourcePortParameters	122
ADQFunctionParameters	123
ADQPatternGeneratorParameters	124
ADQPatternGeneratorInstruction	125
ADQPortParameters	127
ADQPortParametersPin	128
ADQPulseGeneratorParameters	129

ADQSampleSkipParameters	130
ADQSampleSkipParametersChannel	131
ADQSignalProcessingParameters	132
ADQTestPatternParameters	133
ADQTestPatternParametersChannel	133
ADQTestPatternParametersPulse	134
ADQTimestampSynchronizationParameters	134
Status	137
ADQAcquisitionStatus	137
ADQDataReadoutStatus	137
ADQDramStatus	137
ADQOverflowStatus	138
ADQP2pStatus	138
ADQP2pStatusChannel	139
ADQTemperatureStatus	139
ADQTemperatureStatusSensor	140
Data	141
ADQGen4Record	141
ADQGen4RecordHeader	142

A.3.1 Initialization Parameters

This section lists the structures used to configure the digitizer during the initialization phase. See Section 14.3 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 ADQClockReferenceSource 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. 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_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 ADQClockReferenceSource`)

An `ADQClockReferenceSource` that will be used as clock reference for the internal PLL of the digitizer, assuming the `clock_generator` is set to `ADQ_CLOCK_GENERATOR_INTERNAL_PLL`. The default value is `ADQ_CLOCK_REFERENCE_SOURCE_INTERNAL`. Valid values are:

- `ADQ_CLOCK_REFERENCE_SOURCE_INTERNAL`
- `ADQ_CLOCK_REFERENCE_SOURCE_PORT_CLK`

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 clock reference delay adjustment, in units of picoseconds. 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 clock reference 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.

A.3.2 Configuration Parameters

This section lists the structures used to configure the digitizer before initiating the acquisition process. See Section 14.4 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 ADQConstantParameters {
    enum ADQParameterId          id;
    int32_t                      reserved;
    enum ADQClockSystemParameters clock_system;
    int32_t                      nof_channels;
    int32_t                      nof_pattern_generators;
    int32_t                      max_nof_pattern_generator_instructions;
    int32_t                      nof_pulse_generators;
    char                         dna[40];
    char                         serial_number[16];
    char                         product_name[32];
    char                         product_options[32];
    char                         firmware_name[32];
    char                         firmware_revision[16];
    char                         firmware_type[16];
    struct ADQConstantParametersChannel channel[ADQ_MAX_NOF_CHANNELS];
    struct ADQConstantParametersPort port[ADQ_MAX_NOF_PORTS];
    struct ADQConstantParametersCommunicationInterface communication_interface;
    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.

`reserved` (`int32_t`)

Reserved

`clock_system` (`enum ADQClockSystemParameters`)

The parameter set of the currently active clock system configuration.

`nof_channels` (`int32_t`)

The number of physical 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.

`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. [1] [2]

`firmware_name[32]` (`char`)

The firmware name as a zero-terminated array of ASCII characters, i.e. a C-string. For example, "ADQ3-1CH-FWDAQ-PCIE".

`firmware_revision[16]` (`char`)

The firmware revision as a zero-terminated array of ASCII characters, i.e. a C-string. For example, "59000".

`firmware_type[16]` (`char`)

The firmware type as a zero-terminated array of ASCII characters, i.e. a C-string. Designates whether the firmware is a standard firmware ("STANDARD") or built from a development kit ("DEVKIT").

`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.

`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                base_sampling_rate;
    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;
    struct ADQConstantParametersFirFilter fir_filter;
}
  
```

Description

This struct is a member of `ADQConstantParameters` and defines constant parameters for a channel.

Members

`base_sampling_rate (double)`

The base sampling rate in Hz. This value changes depending on the clock configuration. See Section 4 for details.

`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`.

`fir_filter` (`struct ADQConstantParametersFirFilter`)

This struct defines the constant parameters for the FIR filter of the channel.

```
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 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 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_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;
    enum ADQEventSource trigger_source;
    enum ADQEdge      trigger_edge;
    enum ADQFunction  trigger_blocking_source;
    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 current firmware:

- 2CH-FWDAQ: valid range of $[-16360, 2^{35} - 8]$, step size of 8
- 1CH-FWDAQ: valid range of $[-16336, 2^{36} - 16]$, step size of 16

`record_length` ([int64_t](#))

The record length in samples. The default value is 0, and the valid range depends on the current firmware:

- 2CH-FWDAQ: valid range of $[16, 2^{32} - 1]$
- 1CH-FWDAQ: valid range of $[32, 2^{32} - 1]$

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 $[0, 2^{32} - 1]$.

`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_SYNC`
- `ADQ_EVENT_SOURCE_GPIOAO`
- `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 9.3 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.

`reserved` (`int32_t`)

An `ADQEdge` which specifies the edge selection of the `trigger_source`. The default value is `ADQ_EDGE_RISING`.

```

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_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;
    enum ADQMarkerMode marker_mode;
    int32_t      write_lock_enabled;
    int32_t      transfer_records_to_host_enabled;
    int32_t      packed_buffers_enabled;
}
```

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.

`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 10.4). 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 10.3). 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 10.3) 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 10.4) is used.

If the data transfer interface (Section 10.3) 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 10.4) 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 10.3). 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`.

```

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;
    volatile void * record_buffer[ADQ_MAX_NOF_BUFFERS];
    volatile void * metadata_buffer[ADQ_MAX_NOF_BUFFERS];
    volatile void * marker_buffer[ADQ_MAX_NOF_BUFFERS];
    int32_t       record_length_infinite_enabled;
    int32_t       metadata_enabled;
}
  
```

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 should be calculated by the user application as the `record_length` multiplied by the size of a sample, see Section 3. The default value is 0.

`record_buffer_size` (`int64_t`)

The effective record transfer buffer size in bytes. This value must be a multiple of `record_size`. The multiple 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.

`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 void *)`

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.

`record_length_infinite_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.

```

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_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               reserved;
}
```

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).

reserved (int32_t)

Reserved

```
struct ADQDataReadoutParametersChannel {
    int64_t  nof_record_buffers_max;
    int64_t  record_buffer_size_max;
    int64_t  record_buffer_size_increment;
    int32_t  incomplete_records_enabled;
    int32_t  bytes_per_sample;
}
```

Description

This struct is a member of [ADQDataReadoutParameters](#) and defines the data readout parameters for a channel.

Members

nof_record_buffers_max (int64_t)

Reserved

record_buffer_size_max (int64_t)

Reserved

record_buffer_size_increment (int64_t)

Reserved

incomplete_records_enabled ([int32_t](#))

Reserved

bytes_per_sample ([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 ADQDbParametersChannel {
    int64_t level;
    int64_t lower_saturation_level;
    int64_t upper_saturation_level;
    int32_t bypass;
    int32_t reserved;
}
  
```

Description

This struct is a member of [ADQDbParameters](#) 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.

`bypass (int32_t)`

Set to a nonzero value to bypass. The default value is 1.

`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 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 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 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 ADQEventSourceParameters {
    enum ADQParameterId          id;
    int32_t                      reserved;
    struct ADQEventSourcePeriodicParameters periodic;
    struct ADQEventSourceLevelParameters level;
    struct ADQEventSourcePortParameters port[ADQ_MAX_NOF_PORTS];
    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.

`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`.

`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_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 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. 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. 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. 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 base sampling rate 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 ADQEventSourcePortParameters {
    enum ADQParameterId id;
    int32_t             reserved;
    double              threshold;
    uint64_t            magic;
}
```

Description

This struct defines the parameters of the event source associated with a port. See Sections 6.5–6.6

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`

This is guaranteed if `InitializeParameters()` is called to initialize the parameter set.

`reserved` (`int32_t`)

Reserved

`threshold` (`double`)

The threshold in Volts. The default value is 0.5 V.

`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 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];
    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`.

`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 instruc-

tion. 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 sample periods until the next instruction is loaded. The valid range depends on the current firmware:

- 2CH-FWDAQ: valid range of $[8, 2^{35} - 1]$
- 1CH-FWDAQ: valid range of $[16, 2^{36} - 1]$

[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 $1, \dots, 2^{32} - 1$.

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_GPIOAO](#)

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_GPIOAO](#)

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 ADQPortParameters {
    enum ADQParameterId      id;
    int32_t                  reserved;
    struct ADQPortParametersPin pin[ADQ_MAX_NOF_PINS];
    uint64_t                 magic;
}
  
```

Description

This struct 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 7 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_GPIOAO`

`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 current firmware:

- 2CH-FWDAQ: valid range of $-1, 8, 16, \dots, 2^{19} - 8$
- 1CH-FWDAQ: valid range of $-1, 16, 32, \dots, 2^{20} - 16$

`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_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 current firmware:

- 2CH-FWDAQ: 1, 2, 4, 8, 9, 10, ..., $2^{22} - 1$
- 1CH-FWDAQ: 1, 2, 4, 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 ADQFirFilterParameters fir_filter;
    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).

`fir_filter` (`struct ADQFirFilterParameters`)

A `ADQFirFilterParameters` struct representing the parameters of the FIR filter module (see Section 5.4).

`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 ADQTimestampSynchronizationArm 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_GPIAO](#)

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 ADQTimestampSynchronizationArm](#))

Specifies when the timestamp synchronization should be armed and ready to react to events from the selected event [source](#). Valid values are:

- [ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_IMMEDIATE](#)
- [ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_ACQUISITION](#)

The default value is [ADQ_TIMESTAMP_SYNCHRONIZATION_ARM_IMMEDIATE](#).

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 14 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 acquisition 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](#))
Reserved

```
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 [Start-DataAcquisition\(\)](#) is called.

```
struct ADQOverflowStatus {
    int32_t overflow;
    int32_t reserved;
}
```

Description

This structure 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 {
    int nof_sensors;
    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 ([int](#))
 The number of valid entries in [sensor](#).

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.

A.3.4 Data

This section lists the structures used to represent the data transferred by the digitizer. See Section 14 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`.

`data` (`void *`)
A pointer to a memory region holding the record data. The member `size` *must* be set to the size of this region.

❗ 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;
    float      time_unit;
    uint32_t   reserved;
}
```

Header structure contained in an [ADQGen4Record](#).

Members

version_major ([uint8_t](#))

The major version number, i.e. 1 in 1.3.

version_minor ([uint8_t](#))

The minor version number, i.e. 3 in 1.3.

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](#))

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](#). Only valid when the data readout interface is used (see Section [10.4](#)).

- 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 miscellaneous 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)`

Reserved

`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 [10.3](#)) the value will be limited to 16 bits.

! Important

The record number wraps to zero at the maximum value.

`channel (uint8_t)`

The channel from which the record originated.

`data_format (uint8_t)`

The format of the record data:

- 0: 16-bit, 2's complement representation.

`serial_number[10]` (`char`)

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`)

The time between two samples, expressed in time units (`time_unit`).

`time_unit` (`float`)

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.

`reserved` (`uint32_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_MINOR`. See `ADQAPI_ValidateVersion()` for more information about how to implement version control in the user application space.

General	147
<code>ADQAPI_ValidateVersion</code>	147
Identification	148
<code>CreateADQControlUnit</code>	148
<code>ADQControlUnit_EnableErrorTrace</code>	148
<code>ADQControlUnit_ListDevices</code>	149
<code>ADQControlUnit_SetupDevice</code>	149
Parameter Interface	151
<code>InitializeParameters</code>	151
<code>InitializeParametersString</code>	152
<code>InitializeParametersFilename</code>	153
<code>GetParameters</code>	153
<code>GetParametersString</code>	154
<code>GetParametersFilename</code>	155
<code>SetParameters</code>	156
<code>SetParametersString</code>	156
<code>SetParametersFilename</code>	157
<code>ValidateParameters</code>	158
<code>ValidateParametersString</code>	158
<code>ValidateParametersFilename</code>	159
Data Acquisition	160
<code>StartDataAcquisition</code>	160
<code>StopDataAcquisition</code>	160
Data Transfer	162
<code>WaitForP2pBuffers</code>	162
<code>UnlockP2pBuffers</code>	162
Data Readout	164
<code>WaitForRecordBuffer</code>	164
<code>ReturnRecordBuffer</code>	165
Status Monitoring	167
<code>GetStatus</code>	167
Cleanup	168
<code>DeleteADQControlUnit</code>	168
Miscellaneous	169
<code>SWTrig</code>	169
<code>Blink</code>	169
Development Kit	170
<code>ReadUserRegister</code>	170
<code>WriteUserRegister</code>	170

A.4.1 General

[ADQAPI_ValidateVersion](#) 147

```
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

CreateADQControlUnit	148
ADQControlUnit_EnableErrorTrace	148
ADQControlUnit_ListDevices	149
ADQControlUnit_SetupDevice	149

```
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 separate 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 available 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 14.2.

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

InitializeParameters	151
InitializeParametersString	152
InitializeParametersFilename	153
GetParameters	153
GetParametersString	154
GetParametersFilename	155
SetParameters	156
SetParametersString	156
SetParametersFilename	157
ValidateParameters	158
ValidateParametersString	158
ValidateParametersFilename	159

```
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 [14.4](#) 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. 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 14.4 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 14.4 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 [14.4](#) 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. 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 [14.4](#) 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 14.4 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 14.4 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 14.4 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 14.4 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 14.4 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 14.4 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 14.4 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

StartDataAcquisition	160
StopDataAcquisition	160

```
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 criteria in Section [10.4](#) 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 functions other than those marked “⚡ Thread-safe”.

Data transfer

If the configuration fulfills the criteria in Section [10.3](#) 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 [9.3](#)), 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 10.4), 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

WaitForP2pBuffers	162
UnlockP2pBuffers	162

```
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 10.3) and when [marker_mode](#) is set to [ADQ_MARKER_MODE_HOST_MANUAL](#). Refer to Section 10.3.2 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 10.3) 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

WaitForRecordBuffer	164
ReturnRecordBuffer	165

```

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 is the size of the record buffer's data payload in bytes. 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 10.4).

`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. 15). 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.

```
/* 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.

```
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 10.4).

`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. Since the API owns the memory used in the data readout process (see Section [10.4.2](#)), the value of `buffer` is expected to exactly match the values passed to the user application via [WaitForRecordBuffer\(\)](#).

A.4.7 Status Monitoring

[GetStatus](#) 167

```
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.

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](#).

A.4.8 Cleanup

DeleteADQControlUnit	168
--------------------------------	-----

```
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 Miscellaneous

SWTrig	169
Blink	169

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.

A.4.10 Development Kit

ReadUserRegister	170
WriteUserRegister	170

```

int ReadUserRegister(                                     ⚡ Thread-safe
    int      ul_target,
    uint32_t regnum,
    uint32_t *retval
)
  
```

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. [6]

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(                                   ⚡ Thread-safe
    int      ul_target,
    uint32_t regnum,
    uint32_t mask,
    uint32_t data,
    uint32_t *retval
)
  
```

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. [6]

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

ADQ_EOK	172
ADQ_EINVAL	172
ADQ_EAGAIN	172
ADQ_EOVERFLOW	172
ADQ_ENOTREADY	172
ADQ_EINTERRUPTED	173
ADQ_EIO	173
ADQ_EEXTERNAL	173
ADQ_EUNSUPPORTED	173
ADQ_EINTERNAL	173

```
#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.

Worldwide Sales and Technical Support

spdevices.com

Teledyne SP Devices Corporate Headquarters

Teknikringen 6

SE-583 30 Linköping

Sweden

Phone: +46 (0)13 645 0600

Fax: +46 (0)13 991 3044

Email: spd_info@teledyne.com

Copyright © 2021 Teledyne Signal Processing Devices Sweden AB

All rights reserved, including those to reproduce this publication or parts thereof in any form without permission in writing from Teledyne SP Devices.