TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ14-FWPD

User Guide

Author(s): Teledyne SP Devices
Document ID: 18-2074
Classification: Public
Revision: B
Print date: 2023-11-08

Contents

1 Introduction **3**
1.1 Definitions and Abbreviations . . . 3

2 Features **4**
2.1 Specification . . . 4
2.2 Overview . . . 4

3 Getting Started **7**
3.1 SDK Installation . . . 7
3.1.1 Installing the SDK (Windows) . . . 8
3.1.2 Installing the SDK (Linux) . . . 8
3.1.3 Unsupported Software Features . . . 8
3.2 Collecting Data . . . 8

4 Detailed System Description **8**
4.1 Digital Baseline Stabilization . . . 9
4.2 Moving Average . . . 9
4.3 Pulse Specification . . . 9
4.3.1 Edge Windows . . . 11
4.3.2 Baseline Tracking . . . 11
4.4 Trigger Features . . . 11
4.4.1 Timestamp Synchronization . . . 11
4.4.2 Trigger Blocking . . . 13
4.4.3 Synchronization Signals . . . 14
4.5 Coincidence . . . 16
4.5.1 Example . . . 17
4.6 Detection Window . . . 17
4.7 Data Multiplexer . . . 18
4.8 Pulse Characterization . . . 19
4.8.1 Pulse Width . . . 19
4.8.2 Pulse Peak Value . . . 19
4.8.3 Pulse Peak Value Timestamp . . . 19
4.8.4 Overflow . . . 21
4.9 Histograms . . . 21
4.9.1 Overflow . . . 21
4.10 Padding . . . 22
4.11 User Space Record . . . 22
4.11.1 Fixed Length Record . . . 23
4.11.2 Variable Length Record . . . 23
4.12 Data Collection Modes . . . 23
4.12.1 Raw Pulse Data . . . 23
4.12.2 Every *N*th Record . . . 24
4.12.3 Pulse Metadata . . . 24

4.12.4 Raw Pulse Data with Padding 26

5 User Application Example **29**
5.1 Example in C 29
5.1.1 Python Scripts 29

6 Troubleshooting **29**
6.1 Managing License Files 30
6.1.1 Reading the DNA 30
6.1.2 Updating the Digitizer License 30

A Code Listings **30**

1 Introduction

This document is the user guide for the ADQ14 digitizer running the pulse detection firmware (-FWPD) option. Equipped with the pulse detection firmware, the ADQ14 digitizer provides the user with sophisticated tools to identify, collect and analyze *regions of interest* in the data stream in real-time. This relaxes the requirements on the host computer in terms of sustained bandwidth of the device-to-host interface, storage space and offline data parsing.

Release R220420

The pulse peak value timestamp is now measured relative to the first sample in the detection window, rather than relative to the detection window's trigger event truncated to the sampling grid. Due to this truncation, there was a loss of precision when a trigger source capable of sub-sample resolution was used to trigger the detection window, e.g. the TRIG port.

To get the position of a pulse relative to the trigger event, the value must be adjusted with value of the header field `RecordStart`. Refer to Section 4.8.3 for details.

This change only affects the metadata collection mode (mode 1).

Release R200623

An update to the coincidence function was introduced in release R200623. This update is backwards compatible with respect to its feature set and support for the old API remains. However, to fully take advantage of the added functionality, use the new interface described in Section 4.5.

1.1 Definitions and Abbreviations

Table 1 lists the definitions and abbreviations used in this document and provides an explanation for each entry.

Table 1: Definitions and abbreviations used in this document.

Term	Explanation
ADC	Analog-to-digital converter
API	Application programming interface
DBS	Digital baseline stabilization
FWPD	Pulse detection firmware for ADQ14
GSPS	10^9 samples per second
Horizontal parameters	Used to describe parameters and settings which are specified in samples (SI unit *seconds*). For example, the length of a record of data.
LEW	Leading edge window
MSPS	10^6 samples per second
TEW	Trailing edge window
Vertical parameters	Used to describe parameters and settings which are specified in ADC codes (SI unit *Volt*). For example, the level trigger threshold.

2 Features

This section presents the specification of ADQ14-FWPD along with brief descriptions of some of its core features. Detailed information is found in Section 4. Additionally, information pertaining to the ADQ14 platform in general and not this firmware specifically may be found in the ADQ14 manual [1]. However, if the documents give conflicting information, this document takes precedence for devices running this particular firmware option.

2.1 Specification

The specification for the pulse detection firmware is presented in Table 2. For the general specification of the ADQ14 digitizer, please refer to the ADQ14 data sheet [2].

2.2 Overview

Fig. 1 presents a block diagram outlining the main features of the data path. The diagram is drawn for a four-channel device but applies to one- and two-channel devices as well.

- **Input range**

  ADQ14 digitizers with the –VG option offer the possibility to tune the input range of the digitizer.

- **DC offset**

  The analog signal may be subjected to a DC offset. If only pulses with a certain polarity is expected, the user may wish to offset the analog signal to place the baseline close to the maximum or minimum ADC code. This allows pulses with amplitudes that would normally saturate the ADC to be safely collected, thus increasing the dynamic range of the digitizer.

Table 2: Specification for ADQ14-FWPD.

Item	Min	Max	Unit
General			
Pulse width	1	-	Sample
Pulse separation	1	-	Sample
Record length (500 MSPS, 1 GSPS)	4	$2^{32} - 1$	Sample
Record length (2 GSPS)	8	$2^{32} - 1$	Sample
Pulse characterization			
Pulse width [1]	1	$2^{16} - 1$	Sample
Pulse peak timestamp [2]	-	$2^{32} - 1$	Sample
Average pulse rate (1 GSPS)	-	250	10^6 Pulses / s
Average pulse rate (2 GSPS)	-	500	10^6 Pulses / s
Average pulse rate for histograms	-	33	10^6 Pulses / s
Histogram bin value	-	1 048 575	-

[1] See Section 4.8.1
[2] See Section 4.8.3

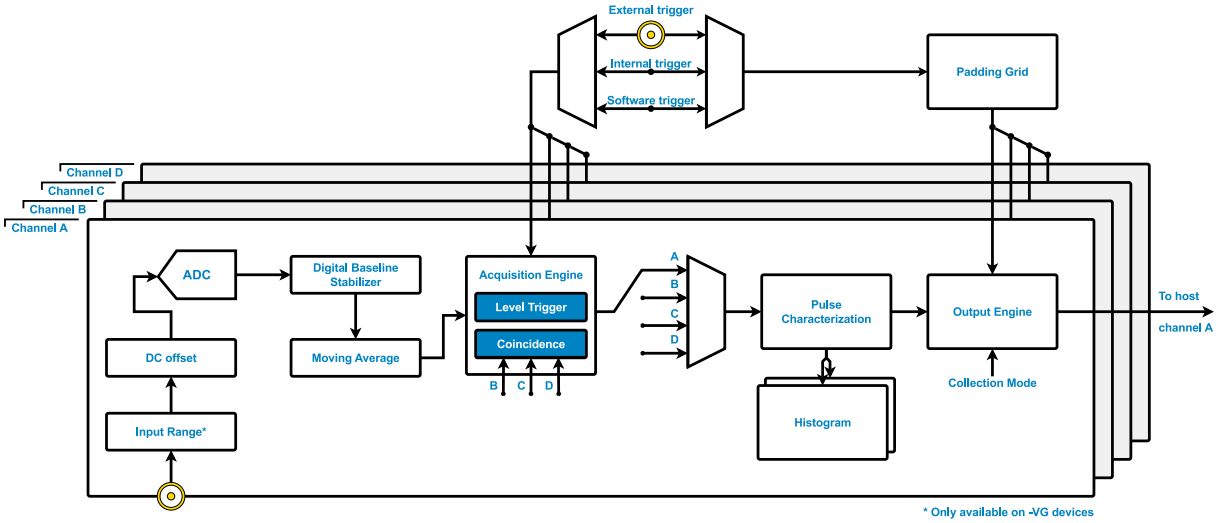


Figure 1: A block diagram presenting the main features of the ADQ14-FWPD data path.

- **ADC**

  The analog signal is sampled at 500 MSPS (–A devices), at 1 GSPS (–C devices) or at 2 GSPS (–X devices).

- **Digital baseline stabilizer**

  The digital baseline stabilizer (DBS) provides a highly accurate and stable baseline, specifically intended for pulse applications. Additional information on the DBS is provided in Section 4.1.

- **Moving average**

  The moving average function tracks the baseline of the signal using a configurable window looking at the last 100 samples. Tracking the baseline allows the user to input the *pulse specification* relative to the baseline rather than in absolute ADC codes. The moving average function is discussed further in Section 4.2.

- **Acquisition engine**

  The acquisition engine is tasked with creating records of data. Each channel is configured independently and requires a trigger condition and a specification of the horizontal parameters.

  - **Trigger**

    The trigger condition may be selected from one of the three global trigger sources: external trigger, internal trigger or software trigger. The latter is mainly used for debugging purposes and has limited practical application. In this case, the selected source is shared among all channels using the global trigger source. Another option is to use the level trigger, in which case each channel operates independently since the trigger condition is detected by analyzing the local data stream. The different ways of triggering a record are discussed throughout Section 4, particularly in Sections 4.4, 4.6 and 4.11.

  - **Coincidence**

    The coincidence function allows the user to impose additional conditions to the trigger mechanism. The downstream propagation of an event on a given channel can be conditioned on the occurence of previous or future events on other channels within a up to 2 us long window. An example of a coincidence condition is provided below.

    Example

    "A trigger on channel A should be accepted and create a record if channel B has detected a trigger in the last 100 ns."

  In addition to the trigger condition and coincidence masking, the definition of a region of interest may be refined further by using a *detection window* (Section 4.6). This window defines a period in time during which pulses are accepted for analysis. Any pulses outside of the window are rejected and are prevented from propagating to the user.

- **Data multiplexer**

  The data multiplexer allows the user to select which data stream to input to the pulse characterization process. In this way, the data collection modes (discussed in Section 4.12) make it possible to verify the system by observing the raw pulse data together with its corresponding metadata. The data multiplexer is discussed in Section 4.7.

- **Pulse characterization**

  The pulse characterization function offers real-time analysis of the pulse data where three key metrics are extracted: the *peak value*, the *width* (time-over-threshold) and the *time of the peak value* relative to the trigger timestamp. The pulse metadata format is described in detail in Section 4.12.3.

  Additionally, the metadata collection mode (discussed in Section 4.12) allows the user to reduce the output data rate further by choosing to only output the pulse metadata.

- **Histogram**

  The pulse peak value and width are forwarded to two independent histograms which are tasked with keeping track of the *frequency of occurrence* of events of a certain type. Details on the histogramming feature may be found in Section 4.9.

- **Padding grid**

  The padding grid defines a window in time, which acts as a guide when the digitizer inserts padding data. This padding data is required in applications which require a certain minimum activity regardless of the event frequency, while still maintaining a high throughput capacity of the device-to-host interface.

  Inserting padding data into the data stream introduces overhead and thus lowers the effective bandwidth of the device-to-host interface (the bandwidth used to transfer useful data). This is an inevitable consequence since the concept of padding targets the trade-off between throughput and latency. The goal is to provide the user with the tools needed to find a balanced compromise between the requirements of the application. The padding mechanism is described in detail in Section 4.10.

- **Output engine**

  The output engine offers flexible control over the flow of data. There are five different data collection modes ranging from only pulse metadata to raw pulse data with padding. The output engine is described in detail in Section 4.12.

3 Getting Started

This section provides information on how to interface with ADQ14 and the pulse detection firmware. Table 3 describes the peripherals and their usage.

Table 3: Peripheral connections on ADQ14.

Connector	Description
TRIG	External trigger input/output
SYNC	Synchronization input/output
A	Channel A input (one, two and four-channel devices)
B	Channel B input (two and four-channel devices)
C	Channel C input (four-channel devices)
D	Channel D input (four-channel devices)
CLK	External reference clock input

3.1 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.

3.1.1 Installing the SDK (Windows)

For Microsoft Windows the SDK is installed by running

```
ADQ-setup_rXXXXX.exe
```

where `XXXXX` 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/
```

3.1.2 Installing the SDK (Linux)

The SDK is supported for several Linux distributions and versions. The required files are all included in

```
ADQ_SDK_linux_rXXXXX.tar.gz
```

where `XXXXX` is the version number. The archive contains the SDK installation files, example code and documentation. The `README` file, located in the root directory of the archive, describes the installation procedure in detail for the different distributions.

3.1.3 Unsupported Software Features

Listed below are the features supported by ADQ14 which are **not** supported by ADQ14-FWPD

- ADCaptureLab
- LabView API

3.2 Collecting Data

The FWPD example in C is provided together with the SDK. For Windows it is located in

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

after installing the SDK, and for Linux in the

```
examples/ADQAPI_FWPD_example
```

directory of the SDK archive. This example illustrates the capabilities of ADQ14-FWPD, and can be used as-is. However, the example is primarily intended to be used as a guide for creating applications tailored to a specific use case.

4 Detailed System Description

This section provides additional details on important system features.

4.1 Digital Baseline Stabilization

For optimal performance in pulse applications, it is advantageous if the baseline of the signal is locked to a specific value. This is the purpose of the Digital Baseline Stabilizer (DBS). The signal processing block is placed directly after the ADC in the data path.

The DBS uses a blind identification process operating in the background. When activated by the user, DBS will constantly monitor and follow variations in the baseline and correct for time-varying behavior such as temperature variations. The baseline is adjusted to the target value with 22-bit precision.

4.2 Moving Average

The purpose of the moving average function is to allow the user to input the pulse specification (discussed in Section 4.3) relative to the baseline rather than in absolute ADC codes.

The moving average function computes an average of samples earlier in time using a memory with capacity to store the previous 100 samples. From this data set, the user can select a range of samples which should be used to compute the moving average value attributed to the *current* sample. The range is specified as a window which has a *length* and an *offset* from the current sample. Both the length and the offset may be given in the range from 0 to 100 samples. However, the sum of the two parameters may not exceed 100 samples.

The refresh rate of the moving average value is tied to the device sample rate. For –A, –C and –X devices the value is updated every 2, 4 and 8 samples, respectively, i.e. every 4 ns.

4.3 Pulse Specification

In order for the system to sift through the constant stream of data, separating regions of interest from uninteresting regions, the user has to tune the system to look for pulses fulfilling a certain criteria—the *pulse specification*.

A pulse is defined by a *trigger level* and three *hysteresis* (offset) values: the *reset hysteresis*, the *trigger arm hysteresis* and the *reset arm hysteresis*. These four values are specified by the user together with a polarity setting, indicating if the target pulse is

- *positive*, i.e. the pulse grows toward *higher* ADC codes from the baseline or
- *negative*, i.e. the pulse grows toward *lower* ADC codes from the baseline.

The arithmetic involved in computing the absolute detection levels of the pulse differs depending on the polarity. Equations (1) and (2) present the relations for positive and negative pulses, respectively.

Positive pulses

$$\begin{aligned} \text{Reset level} &= \text{Trigger level} - \text{Reset hysteresis} \\ \text{Trigger arm level} &= \text{Trigger level} - \text{Trigger arm hysteresis} \\ \text{Reset arm level} &= \text{Reset level} + \text{Reset arm hysteresis} \end{aligned} \tag{1}$$

Negative pulses

$$\begin{aligned} \text{Reset level} &= \text{Trigger level} + \text{Reset hysteresis} \\ \text{Trigger arm level} &= \text{Trigger level} + \text{Trigger arm hysteresis} \\ \text{Reset arm level} &= \text{Reset level} - \text{Reset arm hysteresis} \end{aligned} \tag{2}$$

Moreover, the conditions to generate a trigger or reset event are stated in (3) and (4) for positive and negative pulses, respectively. The data at time instance n is denoted $x[n]$.

Positive pulses

$$\begin{aligned} \text{Trigger event:} \quad x[n] &\geq \text{Trigger level} \\ \text{Reset event:} \quad x[n] &\leq \text{Reset level} \end{aligned} \tag{3}$$

Negative pulses

$$\begin{aligned} \text{Trigger event:} \quad x[n] &\leq \text{Trigger level} \\ \text{Reset event:} \quad x[n] &\geq \text{Reset level} \end{aligned} \tag{4}$$

Fig. 2 places the levels discussed above into the context of a positive pulse. The figure consists of five subfigures:

- **Fig. 2a—the reset level and events**

  The reset level is computed by subtracting the reset hysteresis from the trigger level. When the *rising edge* of the pulse crosses the trigger level a *trigger event* is generated. Conversely, when the *falling edge* of the pulse crosses the reset level a *reset event* is generated. An event always coincides with a sample point and is chosen as the first sample on or beyond the threshold. These two events are the heart of the pulse detection firmware as they frame the region of interest, aiding subsequent functions such as pulse characterization.

- **Fig. 2b—the trigger arm level**

  The trigger arm level is computed by subtracting the trigger arm hysteresis from the trigger level. In order for the rising edge to generate a trigger event, the system has to be *armed*. For a positive pulse, this means that the signal has to visit ADC codes equal to or less than the trigger arm level. Following a trigger event, the system is automatically disarmed until the arming criteria is met once again.

- **Fig. 2c—the reset arm level**

  The arming mechanism for the reset event is identical to that of the trigger event apart from one detail: they are mirror versions of each other. In order for the falling edge of the pulse to generate a reset event, the signal has to visit ADC codes equal to or greater than the reset arm level. Following a reset event, the system is automatically disarmed until the arming criteria is met once again.

- **Figs. 2d and 2e—false events**

  The arming mechanisms are what enables the protection of the data stream against false events. However, this protection hinges on the hysteresis values input by the user to work correctly. As shown in Fig. 2d, badly configured arm levels may have the same effect as not having them at all.

The arming levels should be set *at least* in the same range as the noise to be effective, as shown in Fig. 2e.

4.3.1 Edge Windows

The values discussed in the previous section target the vertical characteristics of a pulse. How the horizontal characteristics of a pulse are treated by the pulse detection firmware are discussed in more detail in Sections 4.11 and 4.12. However, this section will introduce the *leading edge window* (LEW) and the *trailing edge window* (TEW). The names hint at their purpose; the former defines a window in time *before* the trigger event while the latter defines a window *after* the reset event, as shown in Fig. 3. These windows extend the region of interest outward from the pulse.

Note

The leading edge window and trailing edge window are two fixed-length regions that surround the pulse and allow the user to extend the region of interest.

4.3.2 Baseline Tracking

The baseline tracking offered by the moving average function (Section 4.2) allows the user to specify the trigger level relative to the baseline instead of in absolute ADC codes. Since all the other levels are measured relative to the trigger level, a necessary feature is locking the baseline value when a trigger event occurs. This ensures that the relative specification remains constant for the duration of the pulse (longer if the variable length mode is active, see Sections 4.12.1 and 4.11.2). The value remains locked until the end of the trailing edge window, at which point the baseline tracking is resumed.

Note

The baseline is tracked up until the moment of a trigger event, where its current value is locked. The value remains locked until the trailing edge window has been acquired.

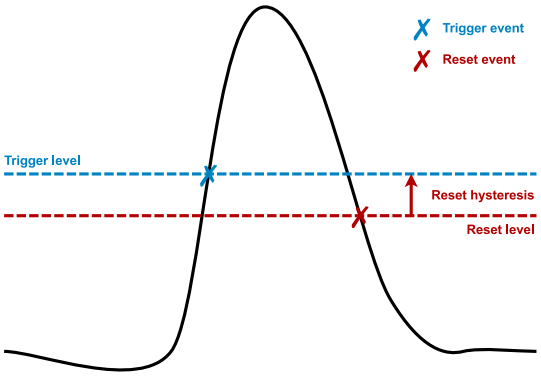
4.4 Trigger Features

This section describes features relating to the external trigger interface: blocking triggers, generating synchronization signals to external equipment and resetting the internal timestamp of the digitizer. Information regarding how a trigger event is used to acquire data is discussed in other sections, namely Sections 4.6, 4.11 and 4.12.

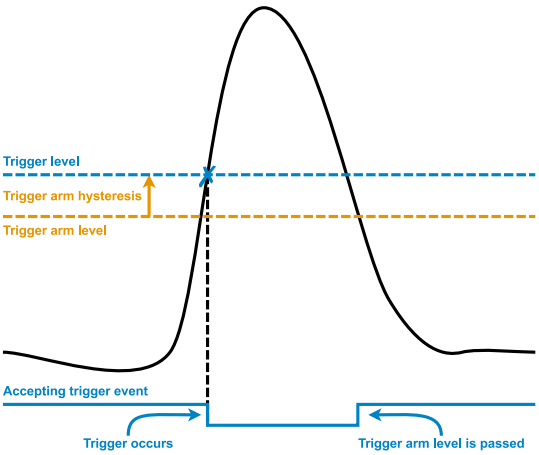
4.4.1 Timestamp Synchronization

The timestamp synchronization feature allows the user to use an external signal to reset the time base of the digitizer. This function is used to establish a common time base across multiple digitizers or to relate the time base to some external event, useful in applications involving some repeated process.

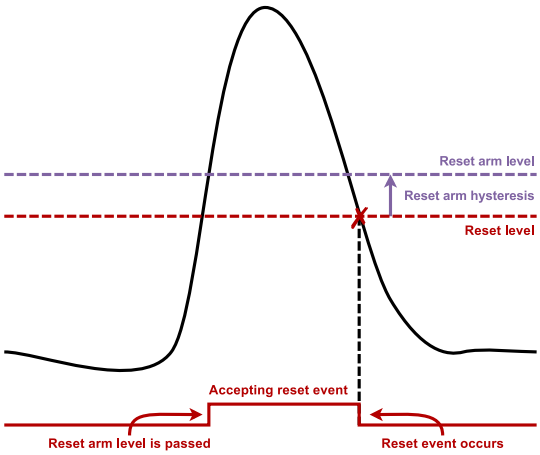
The timestamp synchronization engine listens for external events on either the external trigger connector (TRIG) or the synchronization connector (SYNC). The selected source will be referred to as the



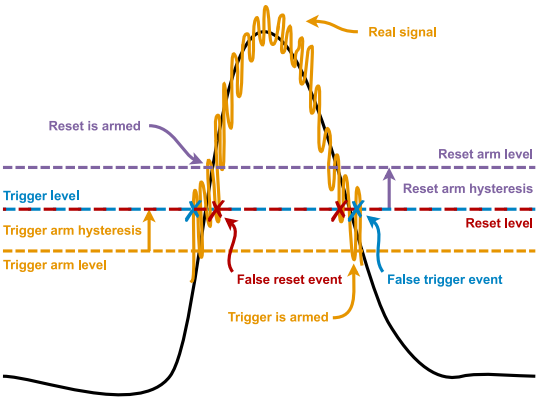
(a) The reset level is determined from the *trigger level* and the *reset hysteresis*.



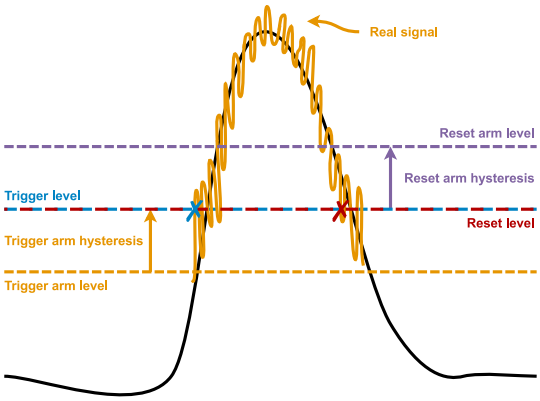
(b) The trigger arm level is determined from the *trigger level* and the *trigger arm hysteresis*. The level is used to arm the mechanism generating trigger events.



(c) The reset arm level is determined from the *reset level* and the *reset arm hysteresis*. The level is used to arm the mechanism generating reset events.



(d) Badly configured arm levels may yield false events from incorrectly detecting pulses within the signal noise.



(e) Setting the arm levels greater than the noise variation protects against false events.

Figure 2: Figs. 2a–2c demonstrate the pulse definition in the context of a positive pulse. Figs. 2d and 2e highlight the purpose of the arm levels.

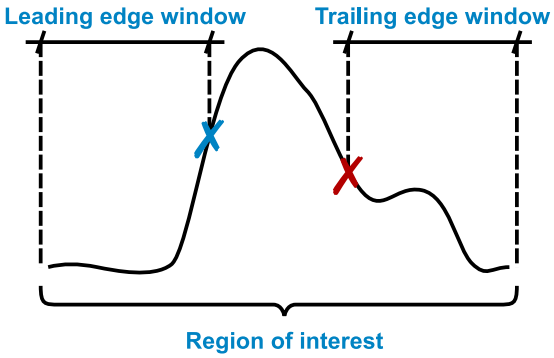


Figure 3: The leading edge window and the trailing edge windows are used to extend the region of interest relative to the trigger event and reset event, respectively.

timestamp synchronization source throughout this section. Each trigger source has an associated edge specification to select if the rising and/or falling edges should generate trigger events. This global edge specification is used by the timestamp synchronization engine and may be set by calling `SetTriggerEdge()`.

Controlling the timestamp synchronization feature involves three functions in the ADQAPI: `SetupTimeStampSync()`, `ArmTimeStampSync()` and `DisarmTimeStampSync()`. The timestamp synchronization feature can be configured in two modes:

First event *Mode index 0*

Once the timestamp synchronization engine is armed, the first event observed on the timestamp synchronization source will reset the timestamp.

Every event *Mode index 1*

Once the timestamp synchronization engine is armed, all subsequent events observed on the timestamp synchronization source will reset the timestamp.

4.4.2 Trigger Blocking

The trigger blocking function generates a blocking window that may be used together with the timestamp synchronization feature to ensure that the digitizer does not create records with unsynchronized timestamps.

The trigger blocking engine listens for external events on either the external trigger connector (TRIG) or the synchronization connector (SYNC). The selected source will be referred to as the *trigger blocking source* throughout this section. The trigger blocking engine uses the global edge specification set by calling `SetTriggerEdge()`.

Controlling the trigger blocking feature involves three functions in the ADQAPI: `SetupTriggerBlocking()`, `ArmTriggerBlocking()` and `DisarmTriggerBlocking()`. The blocking function can be configured in four modes:

Once *Mode index 0*

Once the trigger blocking engine is armed, this mode inhibits the creation of records until the first event on the trigger blocking source has been observed. At this point, the blocking is disengaged and all subsequent triggers are allowed to propagate.

Window *Mode index 1*

Once the trigger blocking engine is armed, this mode inhibits the creation of records until an event on the trigger blocking source has been observed. Following an event, the blocking is disengaged for a fixed duration (the *window length*) and triggers are allowed to propagate. Any additional events on the trigger blocking source that occurs during the window are ignored, i.e. they do not reset the window.

Gate *Mode index 2*

Once the trigger blocking engine is armed, this mode inhibits the creation of records until an event on the trigger blocking source has been observed. Following an event, the blocking is disengaged for as long as the trigger blocking source signal remains 'logic high' or 'logic low' (depending on the source's edge specification) and is reengaged once an event of the opposite type is detected.

Inverse window *Mode index 3*

Once the trigger blocking engine is armed, this mode allows triggers to propagate until an event on the trigger blocking source has been observed. Following an event, the blocking is engaged for a fixed duration (the *window length*), preventing triggers from propagating and creating records during this time.

The trigger blocking feature can be queried by the user for the number of blocking events by calling `GetTriggerBlockingGateCount()`. Additionally, the information is included in the record header and represent the gate counter's value at the time the record was triggered. Combining the timestamp synchronization and trigger blocking features allow the user to keep track of the number of times the timestamp has been reset. In this way, the data from two batches separated by one or several resets of the timestamp are able to be related to each other in time.

Note

The timestamp synchronization and trigger blocking features may be combined to make the gate counter count the number of timestamp resets.

4.4.3 Synchronization Signals

In applications where the digitizer is used to generate the trigger signal to other equipment, the method of choice is to use the internal trigger and output this periodic pulse on the external trigger connector (TRIG). The frame synchronization feature introduces an additional signal synchronized to the internal trigger which may be output on the synchronization connector (SYNC). This signal acts as a frame for the free-running trigger signal and has three configuration parameters: the *frame length*, the *frame factor* and the internal trigger *edge* to use as the reference point for the frame. These parameters are specified by calling the function `SetupFrameSync()`. Additionally, the output has to be activated by calling `EnableFrameSync()`.

The frame generator listens to the internal trigger signal counting the number of edges matching the edge type specified by the user. Once this number is equal to the frame factor, the output is pulled high for a duration equal to the frame's length. Fig. 4 presents an example scenario.

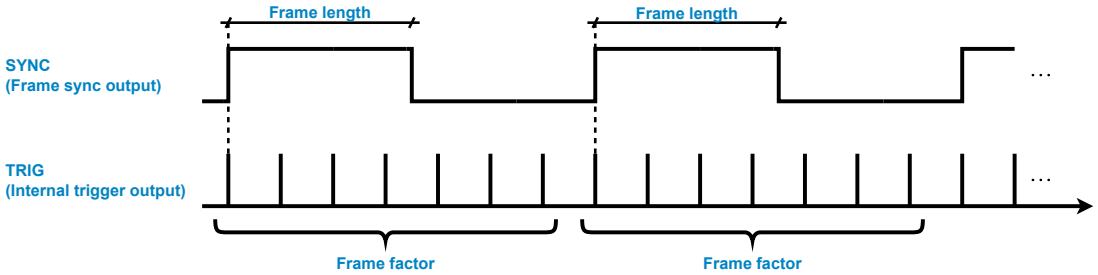


Figure 4: An illustration of the frame synchronization feature. In the depicted scenario, the frame factor is set to 7.

4.5 Coincidence

The coincidence function allows the user to further refine the definition of a region of interest. It offers a mechanism to constrain the conditions of acquiring a pulse by querying the other channels for recent or future events (with some limitations).

Release R200623

In release R200623, this function was upgraded and one of the main additions is the support for non-causal coincidence. However, this comes with a new limitation of the maximum window length which is now 4000 samples.

The function is composed by two building blocks:

- a *core module*, which uses a configurable boolean expression—*the coincidence expression*— together with channel events to generate a gate signal; and
- a *gate matrix* which selects among the available gate signals to mask the events of a specific channel.

There are as many core modules as there are channels on the device, but the modules themselves are not tied to a specific channel. It is the purpose of the gate matrix to tie the output of a core to the masking logic of a specific channel. Fig. 5 presents a block diagram of the function for a four-channel device.

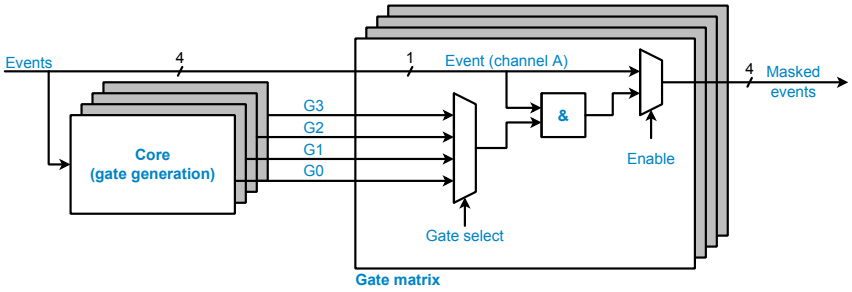


Figure 5: A block diagram of the coincidence function on a four-channel device. There are as many core modules as there are channels. Each channel has a gate matrix which performs the masking for events originating from that channel.

In addition to the coincidence expression, there is another expression which determines when the window is opened: the *start expression*. This expression is fixed logic OR but the operands may be chosen freely among the available channels.

Once the start condition is satisfied, the window is opened and the channels are monitored for events. If the configured coincidence expression is fulfilled for the duration of the window, events on any channel using this expression will be accepted. Otherwise, they will be discarded.

The coincidence expression is implemented as a lookup table. The address used to look up an entry in the table is a binary-encoded value where each bit represents a channel. For each bit, a '1' signifies the presence of an event and '0' the absence of an event within the window. If the expression should ignore one or several channels, these need to be explicitly marked as *don't care* in the table by treating the values '1' and '0' the same. For example, in the expression `B & C`, channels A and D are ignored. This implies that both the presence and absence of events on these channels satisfies the expression.

Thus, the table should encode the same output value for all combinations of '1' and '0' for the ignored channels.

The contents of the lookup table for a few boolean expressions are given below. A hyphen (−) implies a *don't care* condition and `Y` is the output where '1' signifies that the expression is satisfied and '0' that it is not.

`Y = B & C`

D	C	B	A	Y
0	1	1	0	1
0	1	1	1	1
1	1	1	0	1
1	1	1	1	1
All others				0

⇒

`Y = B & C`

D	C	B	A	Y
-	1	1	-	1
All others				0

`Y = (A|B) & C`

D	C	B	A	Y
-	1	0	1	1
-	1	1	0	1
-	1	1	1	1
All others				0

`Y = A & !D`

D	C	B	A	Y
0	-	-	1	1
All others				0

4.5.1 Example

Suppose one of the core modules is configured to use the coincidence expression `B & C` and start expression `B | C` and another core module is configured with the same coincidence expression but instead uses `A | B | C` as its start expression. The implications of the different configurations is presented in Fig. 6. Configuring channel A to use the configuration presented in Fig. 6a will result in pulse A_0 being discarded and pulse A_1 being accepted. If instead the configuration in Fig. 6b is used by channel A, both pulses are accepted. Listing 1 (in the appendix) presents example code in C which demonstrates how to configure the coincidence function for this case.

4.6 Detection Window

A detection window defines a period in time in which pulses are accepted and forwarded to the pulse characterization engine. The detection window has a fixed length and an associated trigger event which may be the output of the global trigger multiplexer or the per-channel level trigger. Fig. 7 demonstrates detection window where two corner cases are highlighted: any pulse not *completely* inside the detection window is rejected. From a technical point of view, *completely* is defined as both the trigger event and the reset event (Section 4.3) occurring inside the window. How the detection window is set up depends on the data collection mode. Refer to Section 4.12 for details.

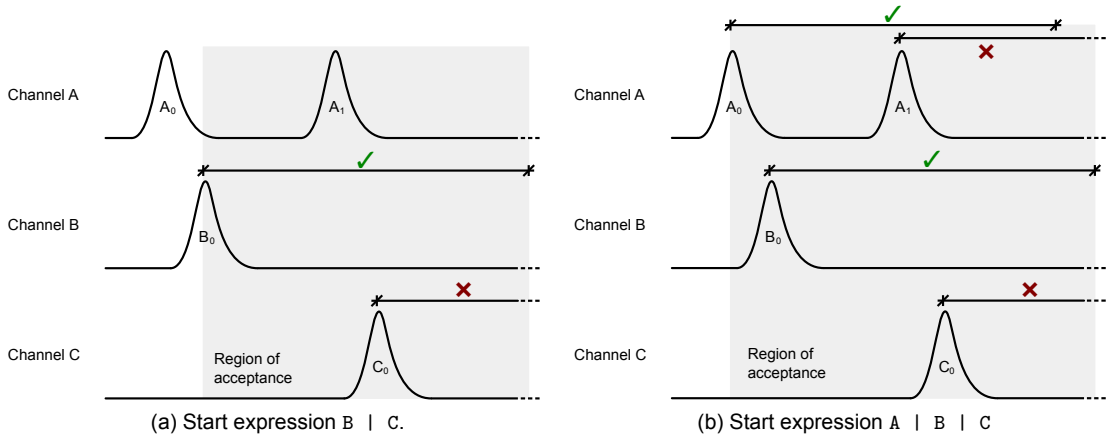


Figure 6: Timing diagram of example 1 (Section 4.5.1). A green check mark highlights the windows during which the coincidence expression `B & C` is satisfied and a red cross highlights the windows where it is not. The region of acceptance is the union of all windows with a green check mark.

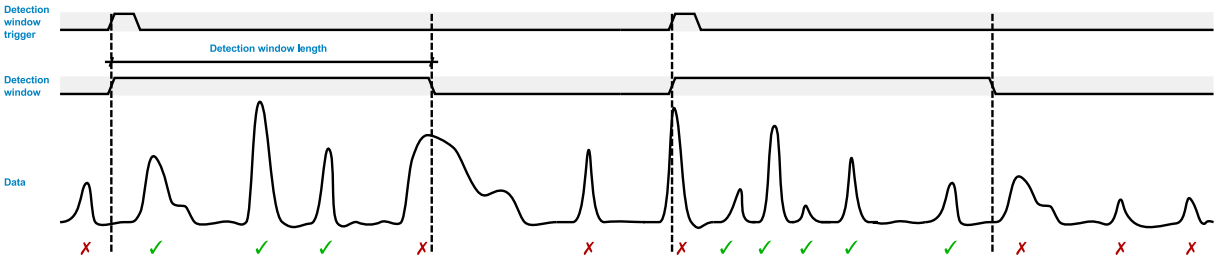


Figure 7: An example of how a detection window is used to mask undesired pulses. Pulses marked with a green check mark are accepted. Pulses marked with a red *x*-mark are rejected.

4.7 Data Multiplexer

The data multiplexer is used to select any of the input data channels and forward the stream of data to any of the characterization channels. The multiplexer allows the user to output a single data stream on multiple channels and may be used to, e.g. verify that the pulse metadata corresponds to the raw pulse data. Each channel of the data multiplexer can be configured to select data from any of the input channels. The multiplexer is controlled with the API call `PDSetDataMux()`

Example

To connect output channel B (channel number 2) to input channel A (channel number 1), issue `PDSetDataMux(1, 2)`

The data multiplexer configuration is reset when the clock configuration is changed, i.e. calling `SetClockSource()` will reset the input channel multiplexer configuration.

4.8 Pulse Characterization

The pulse characterization feature is used to compute key pulse metrics in hardware. The metrics are:

- the pulse width (time-over-threshold),
- the pulse peak timestamp and
- the pulse peak value (extreme value).

These metrics form a pulse metadata packet described in Section 4.12.3. Pulses are characterized during the detection window, as illustrated in Fig. 7 where pulses starting before the window trigger or ending after the window ends are ignored.

4.8.1 Pulse Width

The pulse width is computed as the number of samples between the pulse trigger and reset level. This value will wrap if the pulse width is larger than $2^{16} - 1$ samples.

Note

The pulse width metadata value will wrap if the pulse length is larger than $2^{16} - 1$ samples.

4.8.2 Pulse Peak Value

The maximum value or the minimum value of the pulse. For positive pulses the peak is defined as the maximum value, and for negative pulses the peak is defined as the minimum value. The peak is a 16-bit signed value.

4.8.3 Pulse Peak Value Timestamp

The peak value timestamp is the sample index at which the extreme value occurred, relative to the *first sample* in the detection window. The position of the first sample in the detection window on the timing grid is calculated by adding the record header fields `Timestamp` and `RecordStart`. A common practice is to set the time of the detection window trigger as $t = 0$. In this case, the pulse peak value timestamp only has to be adjusted with the value of `RecordStart`.

The timestamp is a 32-bit unsigned value and will wrap if the distance between the detection window trigger and the pulse is larger than $2^{32} - 1$ samples. If the peak value is reached at multiple samples, the timestamp value will report the last one.

Note

The timestamp value will wrap after $2^{32} - 1$ samples.

Example

Two detection windows, each containing four pulses have been triggered by an external 1 kHz source:

- Window 0:
  - `Timestamp` header value: 34230546665
  - `RecordStart` header value: -137
  - Peak value timestamps: 111, 211, 310, 411
- Window 1:
  - `Timestamp` header value: 34238546738
  - `RecordStart` header value: -146
  - Peak value timestamps: 112, 212, 312, 412

The distance between the detection windows is

$$(34238546738 - 34230546665) \cdot 125 \cdot 10^{-12}\ \text{s} = 1.000009125\ \text{ms},$$

using the fact that the timing grid resolution is 125 ps. This agrees with the 1 kHz trigger frequency.

The position on the timing grid of the first sample in each detection window is calculated by adding the record header fields `Timestamp` and `RecordStart`:

$$t_{first,0} = 34230546665 + (-137) = 34230546528$$

$$t_{first,1} = 34238546738 + (-146) = 34238546592$$

From this, the location of all pulses on the timing grid can be calculated by adding the peak value timestamps, remembering to take the device sampling rate into account.

A common use case is to use the contents of each detection window to build histograms of the pulse metadata. For this purpose, one simplification is to set the detection window trigger point as $t = 0$, i.e. letting $\texttt{Timestamp} = 0$ in the equations above.

The adjusted positions of the peak value timestamps can be computed as

$$t_{peak,adjusted} = \texttt{RecordStart} \cdot 125 \cdot 10^{-12} + t_{peak} \cdot t_s,$$

where t_s is the digitizer's sampling period.

Assuming a sampling period of 1 ns (-C), the adjusted peak value timestamps are:

- Window 0: 93.875 ns, 193.875 ns, 292.875 ns, 393.875 ns
- Window 1: 93.75 ns, 193.75 ns, 293.75 ns, 393.75 ns

4.8.4 Overflow

The pulse characterization module can on average process 500 million pulses / second at 2 GSPS and 250 million pulses / second at 1 GSPS. The module will buffer up to 2048 pulses at 2 GSPS (1024 pulses at 1 GSPS) after which the characterization module overflows. An overflow condition will set a sticky status bit which can be read with the API call `PDGetCharacterizationStatus()`. The status bit is cleared when calling `PDSetupCharacterization()`.

Note

The pulse characterization module can, on average, process 500 million pulses / second at 2 GSPS and 250 million pulses / second at 1 GSPS.

4.9 Histograms

The firmware histogram module provides histograms for the *width* and *peak value* of the pulse. The histograms operation is independent of the data collection mode. The number of histogram bins are listed in Table 4. The histogram bins are represented by a 20-bit unsigned number and will saturate at the maximum value (1 048 575). The data is mapped to a bin number by

$$\text{Bin[n]} = \left\lfloor \frac{(x + \delta) \cdot \alpha}{1024} \right\rfloor, \tag{5}$$

where x is the histogram metadata value, δ is an offset and α is a scale factor. In addition to the normal histogram data bins, each histogram has an overflow and underflow bin. These bins are used if the resulting bin number from (5) is outside of the allowed range. That is if the bin number is smaller than zero the underflow bin will be incremented by one. Conversely, if the bin number is larger than the total number of bins, the overflow bin will be incremented by one. The histogram underflow and overflow bins are accessed by calling `PDGetHistogramStatus()`.

Example

Let a certain pulse have the peak value -5000 and the peak value histogram be configured with scale factor $\alpha = 1024$ and the offset $\delta = 4000$. The resulting bin number would be -1000, from (5). Since this is less than 0, the underflow bin would be incremented by one.

The API calls used to interface with the histograms are listed below. Please refer to the ADQAPI reference guide [3] for details on the usage.

- `PDReadHistogram()`
- `PDReadHistogramStatus()`
- `PDClearHistogram()`
- `PDSetupHistogram()`

4.9.1 Overflow

The average processing bandwidth of the histogram is roughly 33 million pulses / second. For short bursts of pulses, a buffering FIFO is implemented. If the number of pulses exceeds the processing

bandwidth, a overflow flag will be set. The flag can be accessed by calling `ReadHistogramStatus()`. The overflow status is reset by clearing the histograms.

Note

The histogram can process 33 million pulses / second on average. The `ReadHistogramStatus()` API call can be used to detect an overflow.

Table 4: The number of bins for the different histogram types.

Type	Number of bins
Pulse peak value	16384
Pulse width	4096

4.10 Padding

The data padding functionality is used to keep a constant data rate from the digitizer. The padding is active in the data collection modes *pulse metadata* and *raw pulse data with padding*. The padding block will append zeros to the data to ensure that the total amount of data measured over one padding frame is at least equal to the *minimum frame length*. Setting the minimum frame length to zero disables the padding.

Note

The padding functionality is disabled by setting the minimum record length to zero.

The padding is applied to each *padding frame* after a certain offset. The padding offset is equal to the detection window length in pulse metadata mode. The padding is configured using the API call `PDSetupCharacterization()`. How the padding frame is set up depends on the data collection mode, see Table 6. For pulse metadata mode the padding data is appended to the data record, whereas for the raw pulse data with padding mode a new record is generated with the padding data.

4.11 User Space Record

User space records are the records of data received by the user. A record consists of a data part and a header part. The header includes information about e.g. the length and the trigger timestamp. The contents of the user space records will depend on the data collection mode. In all modes but the metadata mode, the record will consist of the data sampled by the ADC. In the metadata mode, the record will consist of the pulse metadata packets. The raw data records can either have a *fixed length* or a *variable length*. Fig. 8 illustrates the differences between the two record types. The length of a metadata record will depend of the number of pulses detected.

4.11.1 Fixed Length Record

A fixed length record is defined by the trigger event and the record length. The length of a fixed length record is specified by the user, and is independent of the data collected. A fixed length record can be triggered by any of the trigger sources.

4.11.2 Variable Length Record

The length of a variable length record will depend on the data. A variable length record is defined as the data between the trigger event and reset event. If trailing or leading edge windows are used, these will also be included in the record.

If two pulses have overlapping trailing and leading edge windows, they will be concatenated into one single record. This is illustrated in Fig. 8. The variable length mode can only be used with the level trigger.

4.12 Data Collection Modes

This section describes the different data collection modes. The collection mode is configured using the API function `PDSetupCharacterization()`. The API function `PDSetupCharacterization()` should only be called after `PDSetupTiming()`. The collection mode determines how the arguments of `PDSetupCharacterization()` are used. This is presented in Table 6. The collection modes are listed in Table 5.

Table 5: This table lists the available data collection modes and their interaction with the padding mechanism and the detection window.

Mode	Description	Padding	Detection window	Section
0	Raw pulse data	No	No	4.12.1
1	Pulse metadata	Yes	Yes	4.12.3
2	Every *N*th record	No	No	4.12.2
3	Raw pulse data	Yes	Yes	4.12.4
4	Raw pulse data	Yes	No	4.12.4

Important

`PDSetupCharacterization()` should only be called after `PDSetupTiming()`.

4.12.1 Raw Pulse Data

This is the default data mode. The raw pulse data mode can be used with any trigger if fixed length records are used, and with the level trigger if variable length records are used. The raw pulse data mode does not insert any padding. In raw pulse data mode the data received is the data sampled by the ADCs. Fig. 8 illustrates the raw pulse mode in both fixed length and variable length mode using the level trigger.

Table 6: This table lists how the arguments of `PDSetupCharacterization()` are used to control the pulse detection features.

Mode	Padding frame trigger	Record trigger	Padding offset
0	N/A	`trigger_mode`	N/A
1	`padding_trigger_mode`[1]	`padding_trigger_mode`	`detection_window_length`
2	N/A	`trigger_mode`	N/A
3	`padding_trigger_mode`[1]	`trigger_mode`	`detection_window_length`
4	`padding_trigger_mode`	`trigger_mode`	`padding_offset`

[1] The padding trigger is also used as the detection window trigger in these modes.

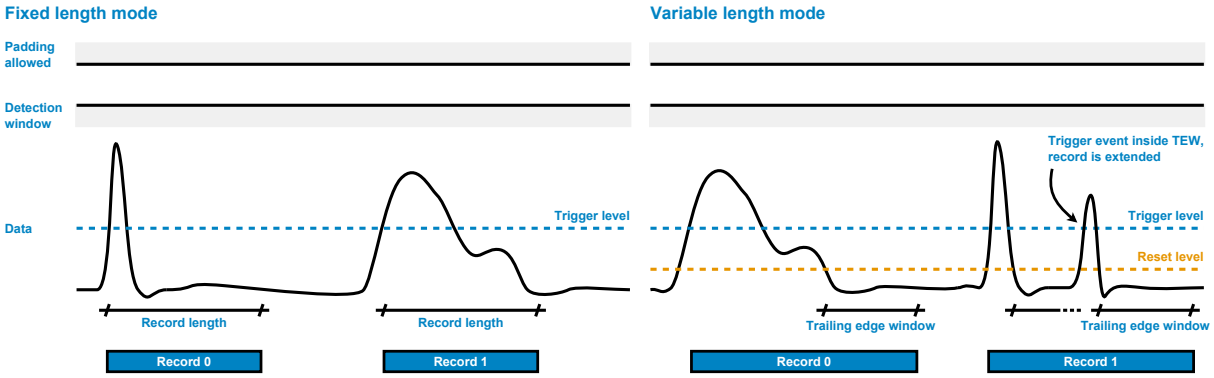


Figure 8: Data collection mode 0 (raw pulse data).

4.12.2 Every *N*th Record

The every *N*th record mode behaves in the same way as the raw pulse data mode with the exception that only every *N*th record is transferred to the host. The reduction factor *N* is specified in the API call `PDSetupCharacterization()` and is *only* used in this mode.

Example

If the reduction factor is set to 5, every 5th record will be transferred to the host.

4.12.3 Pulse Metadata

In the pulse metadata mode, only the pulse metadata is transferred to the host. The individual metadata fields are described in Section 4.8 and Fig. 9 presents how the mode acts on a stream of pulse data with a detection window. The detection window trigger will generate a record which is populated with metadata packets. If no pulses are detected during a detection window an empty metadata data packet will be transferred to the host.

Table 7 describes the layout of the metadata packets. The byte column refers to the position in the returned array. One record can consist of one or several metadata packets. The size of each metadata packet is 8 bytes. Packets used for padding have all fields equal to zero and should be discarded.

Table 8 describes the format of the returned data for each record in pulse characterization mode.

Note

Invalid packets are identified by all fields being equal to zero.

The pulse metadata mode will pad the record with zero-filled metadata packets if the total amount of data during one detection window is less than the specified minimum frame length.

Table 7: The metadata packet structure.

Field	Byte	Size	Format
Pulse peak timestamp	3:0	4 bytes	32-bits unsigned integer
Pulse peak value	5:4	2 bytes	16-bits signed integer
Pulse width	7:6	2 bytes	16-bits unsigned integer

Table 8: This table describes how the contents of a pulse metadata record is structured.

Byte	Description	Comment
7:0	Metadata packet 0	Analysis result of the first pulse in the detection window
15:8	Metadata packet 1	Analysis result of the second pulse in the detection window
23:16	Metadata packet 2	Analysis result of the third pulse in the detection window
…	…	…

4.12.4 Raw Pulse Data with Padding

The pulse data with padding mode will output padding records if the amount of data collected is less than the minimum frame length. The padding records will have all data equal to zero and the LSB of the *User ID* header field set to 1. The pulse data with padding can either be configured to discard (mode 3) or to keep (mode 4) data outside of the detection window. Fig. 10 illustrates the two different modes. The padding record timestamp and record number is undefined and should not be used.

Example

If the minimum frame length is set to 1024 samples and only 800 samples of data are captured, an additional record with 224 samples will be transmitted to the host computer.

Warning

The timestamp and record number for the padding record is undefined.

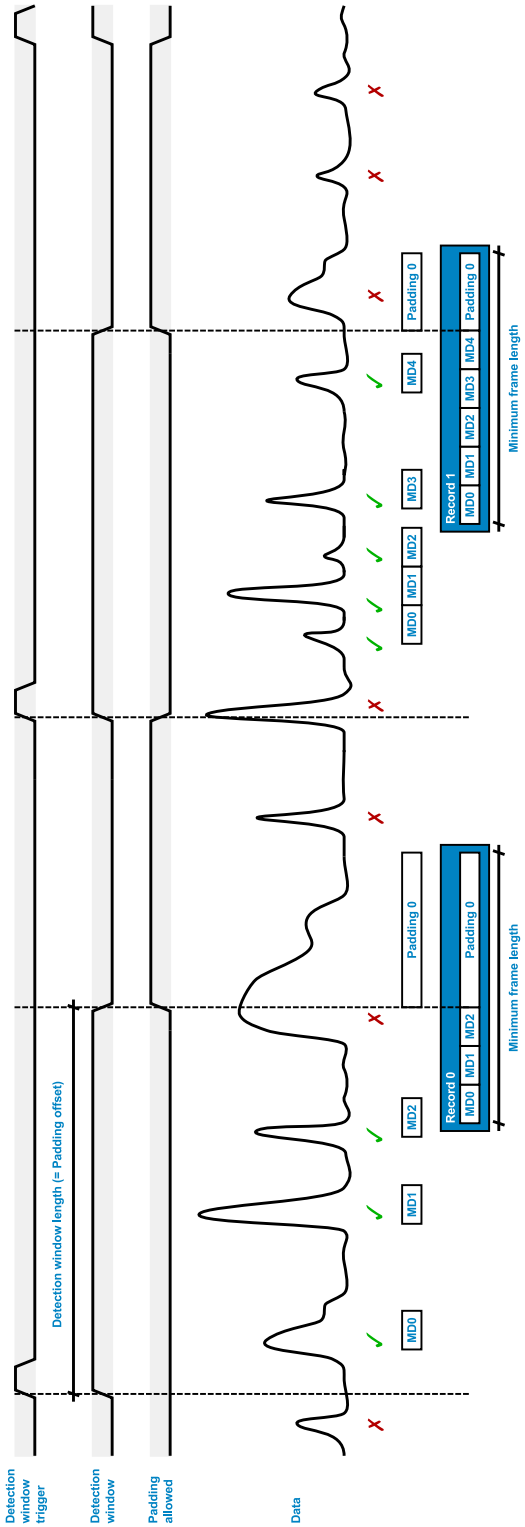
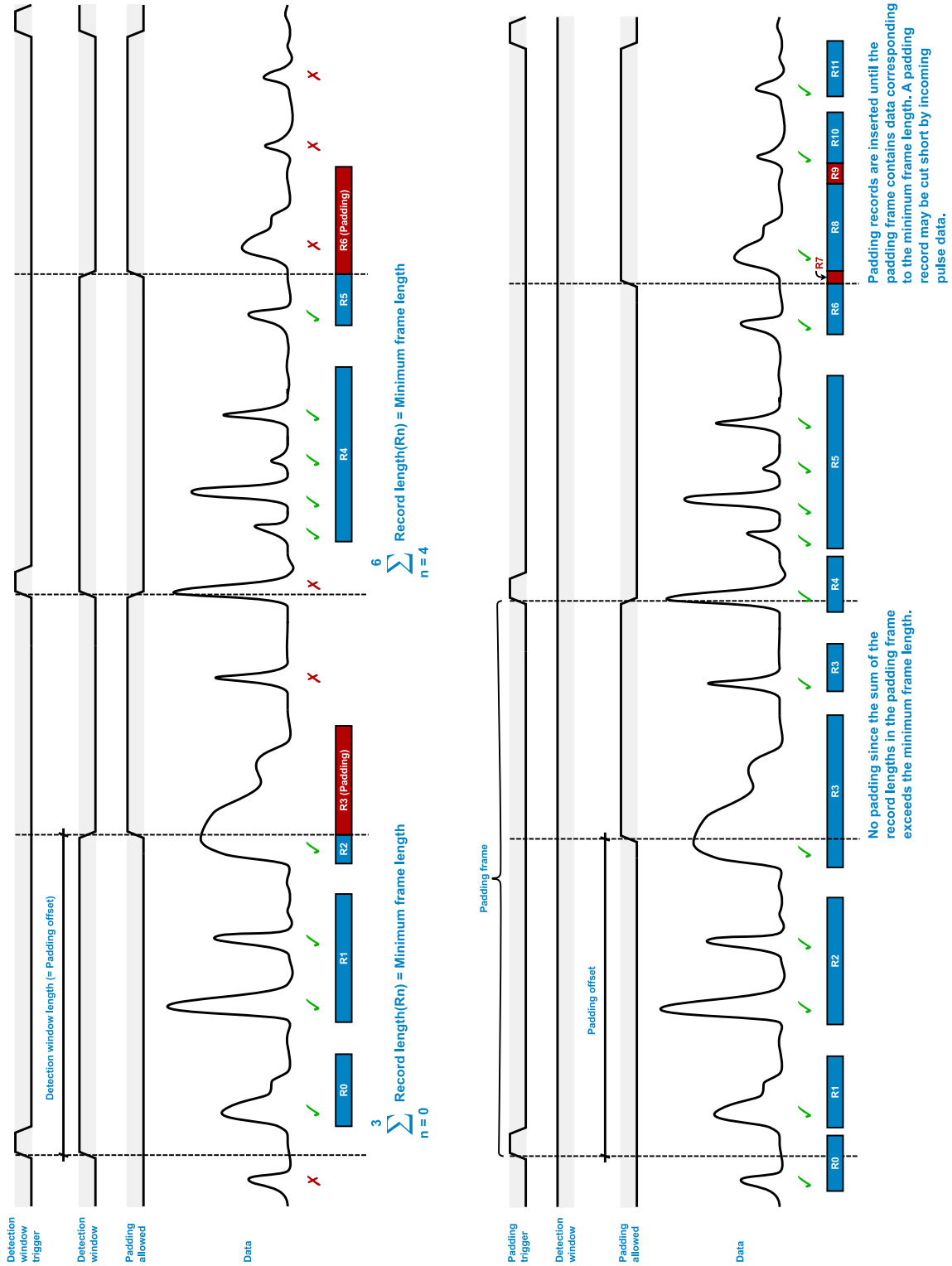


Figure 9: Data collection mode 1 (pulse metadata).



(a) Data collection mode 3 (with detection window).

(b) Data collection mode 4 (without detection window).

Figure 10: Timing diagrams presenting the collection modes 3 and 4 (raw pulse data with padding).

5 User Application Example

This section aims to explain the example code provided for ADQ14-FWPD. The ADQAPI reference guide [3] lists all the functions available in the ADQAPI and is a good supplementary source of information. Please note that not all functions are valid for ADQ14.

5.1 Example in C

An example in C is included in the release archive and located in the `examples/` directory. The example application consists of four files listed in the hierarchy structure below.

`examples/c/fwpd/`

`ADQAPI_FWPD_example.c`	Main example file, digitizer configuration and data collection.
`formatter.h`	Definition of console output formatting functions.
`formatter.c`	Implementation of console output formatting functions.
`streaming_header.h`	Definition of the ADQ14 record header for triggered streaming.

The example consists of a single `main()` function with the configuration parameters listed at the top. The example is verbose by design and is intended as reference for deigning custom applications. The example will configure the device, acquire all the data, save the data to file if instructed to do so and finally deallocate the memory and delete the ADQ object. If the `save_data_to_file` variable is set, the data and header information will be saved to files in the current working directory.

5.1.1 Python Scripts

Two Python scripts are included: `plot.py` and `parse_metadata.py`. The scripts require Python 3, Numpy and Matplotlib. The `plot.py` script can be used for plotting raw pulse data and the `parse_metadata.py` script can be used to parse and print metadata records.

6 Troubleshooting

This section aims to provide some guidance when troubleshooting unexpected behavior. It is recommended that the user application is written in a robust manner, able to capture and report error codes from failed ADQAPI function calls. In the event of a function call failure, reading the ADQAPI trace log for additional information is a useful first step. Trace logging must be activated by calling `ADQControlUnit_EnableErrorTrace()` with the `trace_level` argument set to 3.

If the error message is difficult to interpret, the Teledyne SP Devices support can be reached via e-mail at spd_support@teledyne.com. Please include information about your use case such as the pulse detection settings as well as the specification for both the trigger and data signals. Make sure to include a trace log file from a run where the error appears.

6.1 Managing License Files

ADQ14-FWPD requires a valid license to be able to operate properly. An unsuccessful license validation will result a trace log error message and failure to use certain functions in the ADQAPI. The digitizer licenses are managed with the ADQLicenseUtil tool, included in the SDK. Please direct all license related questions at spd_support@teledyne.com.

6.1.1 Reading the DNA

Licenses are locked to each individual digitizer by means of the *DNA*, a unique 128-bit number. Performing field updates to the digitizer licenses may require the DNA and digitizer serial number to be read out and forwarded to Teledyne SP Devices. This task can be accomplished by opening a command prompt or terminal window with access to the ADQLicenseUtil application and running

```
$ adqlicenseutil d
```

Make a note of the digitizer serial number and DNA in the resulting output text.

6.1.2 Updating the Digitizer License

Provided a valid license file, `<license>.lic`, the command

```
$ adqlicenseutil w <license>.lic
```

transfers the license to the digitizer.

References

[1] Teledyne Signal Processing Devices Sweden AB, *15-1593 ADQ14 manual*. Technical Manual.

[2] Teledyne Signal Processing Devices Sweden AB, *14-1290 ADQ14 datasheet*. Technical Specification.

[3] Teledyne Signal Processing Devices Sweden AB, *14-1351 ADQAPI Reference Guide*. Technical Manual.

A Code Listings

Listing 1: Example code in C to configure the coincidence function as described in Section 4.5.1.

```
/* Allocate an array of 2^nof_channels bytes to represent the lookup table of
   the coincidence expression. The entire array is initialized to zero. */
unsigned char *coincidence_expression =
  calloc(1 << nof_channels, sizeof(unsigned char));

/* Configure coincidence expression B & C. Channels A and D are marked as
   "don't care". */
```

```
unsigned int window_length = 1000;
                                /* DCBA */
coincidence_expression[6] = 1;  /* 0110 */
coincidence_expression[7] = 1;  /* 0111 */
coincidence_expression[14] = 1; /* 1110 */
coincidence_expression[15] = 1; /* 1111 */

int success_core0 =
  ADQ_PDSetupTriggerCoincidenceCore(adq_cu, adq_num, 0,
                                    window_length,
                                    coincidence_expression, 0x6u);
int success_core1 =
  ADQ_PDSetupTriggerCoincidenceCore(adq_cu, adq_num, 1,
                                    window_length,
                                    coincidence_expression, 0x7u);
free(coincidence_expression);

if (!success_core0 || !success_core1)
{
  /* Handle an unsuccessful call. */
}

/* Connect coincidence core 0 to channel A and core 1 to channel B
   and enable the masking logic. The other channels bypass the
   coincidence function. */
unsigned int enable[4] = {1, 1, 0, 0};
unsigned int core[4] = {0, 1, 0, 0};
for (unsigned int ch = 0; ch < nof_channels; ++ch)
{
  if (!ADQ_PDSetupTriggerCoincidence2(adq_cu, adq_num, ch + 1,
                                      core[i], enable[i]))
  {
    /* Handle an unsuccessful call. */
  }
}

/* Reset the coincidence function before continuing with the remaining
   setup. */
if (!ADQ_PDResetTriggerCoincidence())
{
  /* Handle an unsuccessful call. */
}
```

TELEDYNE SP DEVICES
Everywhereyoulook™

Worldwide Sales and Technical Support
spdevices.com

Teledyne SP Devices Corporate Headquarters
Teknikringen 8D
SE-583 30 Linköping
Sweden
Phone: +46 (0)13 645 0600
Fax: +46 (0)13 991 3044
Email: spd_info@teledyne.com

Copyright © 2023 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.