ADQGen3 Streaming

User Guide

Author(s): Teledyne SP Devices
Document ID: 20-2465
Classification: Public
Revision: B
Print date: 2021-05-04

Contents

1 Introduction **2**
- 1.1 Definitions and Abbreviations . . . 2
- 1.2 Overview . . . 3

2 Data Acquisition **3**
- 2.1 Configuration . . . 4

3 Data Transfer and Data Readout **6**
- 3.1 Configuration . . . 6
  - 3.1.1 Data Transfer . . . 6
  - 3.1.2 Data Readout . . . 6
- 3.2 Transfer Buffers . . . 7
- 3.3 Interface . . . 8
- 3.4 Record Buffers . . . 8
- 3.5 Program Flowcharts . . . 9
- 3.6 Incomplete Records . . . 13
- 3.7 Status Events . . . 13
- 3.8 Overflow . . . 13
  - 3.8.1 Physical Interface (case 1) . . . 13
  - 3.8.2 Readout Interface (case 2) . . . 14

A API Reference **16**
- A.1 Enumerations . . . 16
- A.2 Structures . . . 18
- A.3 Configuration Functions . . . 29
- A.4 Data Readout Functions . . . 33

Document History

Revision	Date	Section	Description	Author
B	2021-05-04	-	Make the document valid for ADQ7 and ADQ14	TSPD
A	2020-10-15	-	Initial release	TSPD

1 Introduction

This document is a user guide for continuous data acquisition and transfer to a host computer, i.e. *streaming*. How to configure the digitizer's other supporting functions, e.g. timestamp synchronization or various signal processing modules is not described in this document. Refer to the ADQAPI reference guide [1] for documentation of functions not included in this document.

Important

This document is only valid for the following digitizer models:

- ADQ8
- ADQ7 **(requires SDK revision 58988 or later)**
- ADQ14 **(requires SDK revision 58988 or later)**

1.1 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
API	Application programming interface
Development kit	Toolchain to build customized firmware (FPGA contents) for a digitizer
DMA	Direct memory access
Horizontal offset	The offset (in samples) between the trigger event and the first sample in the record
I/O	Input/output
MiB	Mebibyte, i.e. $1024 \cdot 1024$ bytes.
Physical interface	The device-to-host interface, e.g. USB or 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, examples and documentation)
SSD	Solid-state drive
Streaming	The act of continuously acquiring and propagating data to an application running on a host computer
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.

1.2 Overview

The act of continuously acquiring and propagating data to an application running on a host computer is called *streaming*. This concept consists of three parts: data acquisition, data transfer and data readout.

- Section 2 presents the data acquisition process.
- Section 3 presents the data transfer and data readout processes.
- Appendix A lists all the relevant API functions to accomplish this task.

2 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, e.g. USB or PCIe.

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

The acquisition parameters for each digitizer channel is independently configurable in terms of

- the number of records to acquire,
- the record length,
- the trigger event source and its edge; and
- the horizontal offset.

Fig. 1 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.

(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.

The digitizer has a built-in time-keeping mechanism with a certain timing resolution, e.g. 25 ps. At power-on, a counter starts to monotonically increment—creating a timing grid (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 `RecordStart`.

Note that the timing resolution varies between event sources. For example, a level trigger event will have sample resolution while an external event detected on the TRIG port will have 250 ps resolution.

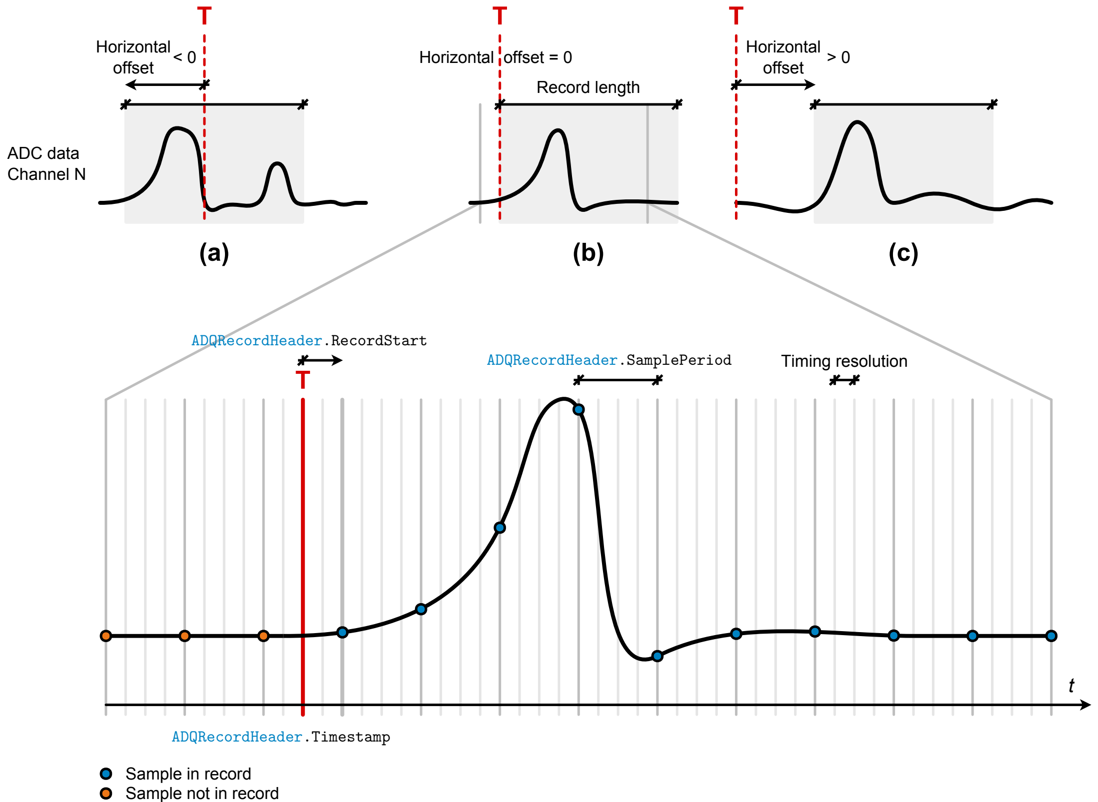

Figure 1: Illustration of how the parameters of the data acquisition process affect the acquisition of a record. Relative values, e.g. the record header field `RecordStart`, are all anchored in the trigger event **T**. The header field `Timestamp` specifies where the trigger event is located on the timing grid.

> **Important**
>
> The timing resolution varies between event sources.

Once the parameters have been set, the acquisition process is started by calling `StartDataAcquisition()`. 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.

2.1 Configuration

The data acquisition process is configured by passing an `ADQDataAcquisitionParameters` struct to `SetParameters()`. The parameter set should be seeded with default values by calling `InitializeParameters()`. Refer to Appendix A.2 for a detailed description of the available parameters.

The following code uses the C API to demonstrate how to initialize the data acquisition parameters to their default values and configure the first channel to acquire 10 records, each with 2048 samples taken at the rising edge of the signal input on the TRIG port. A channel is activated by specifying a nonzero number of records for that channel (the default value is zero).

```
/* Set up the data acquisition process. */
struct ADQDataAcquisitionParameters acquisition_parameters;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_DATA_ACQUISITION,
                                      &acquisition_parameters);
if (result != sizeof(acquisition_parameters))
{
  /* Handle error (see Appendix A) */
}

/* Activate one of the channels. */
acquisition_parameters.channel[0].nof_records = 10;
acquisition_parameters.channel[0].record_length = 2048;
acquisition_parameters.channel[0].trigger_source = ADQ_EVENT_SOURCE_TRIG;
acquisition_parameters.channel[0].trigger_edge = ADQ_EDGE_RISING;

/* The other parameters get their default values from InitializeParameters(). */

result = ADQ_SetParameters(adq_cu, adq_num, &acquisition_parameters)
if (result != sizeof(acquisition_parameters))
{
  /* Handle error (see Appendix A) */
}
```

It is also possible to update the value of a single parameter by performing a read-modify-write operation using `GetParameters()`, demonstrated by the following code snippet.

```
/* Read the current parameters of the data acquisition process. */
struct ADQDataAcquisitionParameters acquisition_parameters;
int result = ADQ_GetParameters(adq_cu, adq_num,
                               ADQ_PARAMETER_ID_DATA_ACQUISITION,
                               &acquisition_parameters);
if (result != sizeof(struct ADQDataAcquisitionParameters))
{
  /* Handle error (see Appendix A) */
}

/* Change to an infinite stream of records. */
acquisition_parameters.channel[0].nof_records = ADQ_INFINITE_NOF_RECORDS;

/* The other parameters keep their values from GetParameters(). */

result = ADQ_SetParameters(adq_cu, adq_num, &acquisition_parameters)
if (result != sizeof(struct ADQDataAcquisitionParameters))
{
  /* Handle error (see Appendix A) */
}
```

3 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 host computer's RAM. Once data is available in these buffers, the data readout process performs the necessary steps to parse the transferred data into records that are passed to the user application.

These two processes are 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. Fig. 2 presents an overview and the following sections describe the process in more detail.

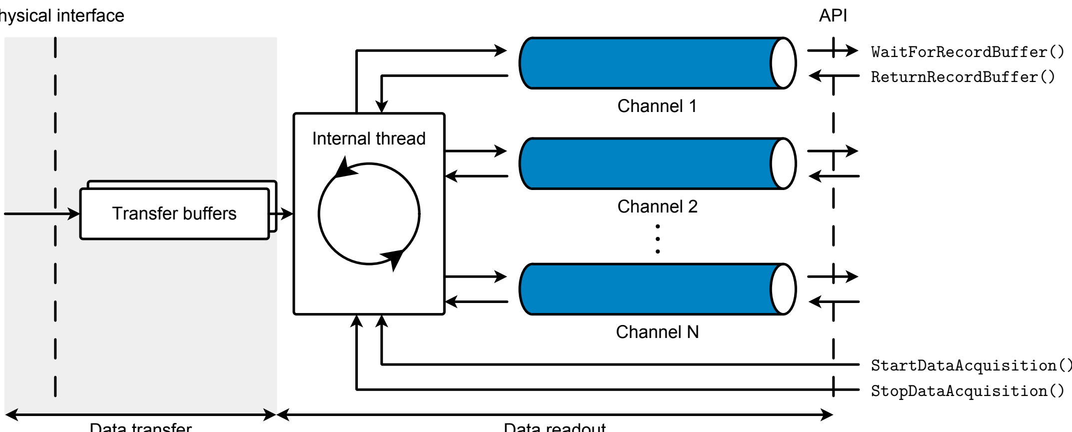

Figure 2: Overview of the data transfer and data readout processes. An internal thread manages the transfer of data between the digitizer and the host computer. Records are passed to the user via thread-safe channels corresponding to channels on the digitizer.

3.1 Configuration

This sections describes how to configure the data transfer and data readout processes.

3.1.1 Data Transfer

The parameters of the data transfer process is configured by passing an `ADQDataTransferParameters` struct to `SetParameters()`. The parameter set should be seeded with default values by calling `InitializeParameters()`. Refer to Appendix A.2 for a detailed description of the available parameters.

3.1.2 Data Readout

The parameters of the data transfer process is configured by passing an `ADQDataReadoutParameters` struct to `SetParameters()`. The parameter set should be seeded with default values by calling `InitializeParameters()`. Refer to Appendix A.2 for a detailed description of the available parameters.

The following code uses the C API to demonstrate how to initialize the data readout parameters to their default values and configure the parameter bounding the memory consumption of the first channel.

```
/* Set up the data readout process. */
struct ADQDataReadoutParameters readout_parameters;
int result = ADQ_InitializeParameters(adq_cu, adq_num,
                                      ADQ_PARAMETER_ID_DATA_READOUT,
                                      &readout_parameters);
if (result != sizeof(readout_parameters))
{
  /* Handle error (see Appendix A) */
}

/* Tune the parameters bounding the memory consumption of the first channel. */
readout_parameters.channel[0].nof_record_buffers_max = 64;
readout_parameters.channel[0].record_buffer_size_max = 16 * 1024 * 1024;
readout_parameters.channel[0].record_buffer_size_increment = 2 * 1024 * 1024;

/* The other parameters get their default values from InitializeParameters(). */

result = ADQ_SetParameters(adq_cu, adq_num, &readout_parameters)
if (result != sizeof(readout_parameters))
{
  /* Handle error (see Appendix A) */
}
```

It is also possible to update the value of a single parameter by performing a read-modify-write operation using `GetParameters()`, demonstrated by the following code snippet.

```
/* Set up the data readout process. */
struct ADQDataReadoutParameters readout_parameters;
int result = ADQ_GetParameters(adq_cu, adq_num,
                               ADQ_PARAMETER_ID_DATA_READOUT,
                               &readout_parameters);
if (result != sizeof(readout_parameters))
{
  /* Handle error (see Appendix A) */
}

/* Increase the number of buffers in rotation for the first channel. */
readout_parameters.channel[0].nof_record_buffers_max = 200;

/* The other parameters keep their values from GetParameters(). */

result = ADQ_SetParameters(adq_cu, adq_num, &readout_parameters)
if (result != sizeof(readout_parameters))
{
  /* Handle error (see Appendix A) */
}
```

3.2 Transfer Buffers

Before the data is sent over the physical interface, it is serialized into a single stream. This means that the data is *packed* before leaving the digitizer and that the host computer has to *mirror* this operation

to split this single stream into separate streams for each channel. This parsing involves a memory copy operation which is carried out by the internal thread as it continuously parses the contents of a transfer buffer into record buffers for the appropriate channel (see Fig. 2).

The packed data is transferred with direct memory access (DMA) to transfer buffers, which are dedicated memory regions in the host computer's RAM. These are allocated by the API, however it is possible to specify the size and number of these buffers via the data transfer parameters `nof_buffers` and `record_buffer_size`.

The size of a transfer buffer affects the behavior of the data readout interface as a trade-off between latency (smaller buffers) and throughput (larger buffers). The optimal value depends on the physical interface type and the behavior expected by the user application. Additionally, the number of transfer buffers may affect the throughput. At least two are required, however if the system can spare the memory resources, a higher number is recommended.

Tuning the transfer buffer parameters is an expected step when integrating the digitizer into a target application. The default size is set in the MiB-range to achieve high throughput.

> **Note**
> The memory regions used as transfer buffers may be subject to platform-dependent requirements. For example, on Linux systems, there may be an upper bound of 16 MiB placed on the total size of the transfer buffers. This upper limit can be changed by modifying the appropriate kernel parameters.

3.3 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 Appendix A.4 for detailed descriptions of these functions.

3.4 Record Buffers

As a transfer buffer is parsed (Section 3.2) and records are constructed, the data is copied to *record buffers*. These are memory regions with the specific purpose of holding the record data and its associated metadata, i.e. its *header*. The expected memory format of a record buffer is specified by the `ADQRecord` struct. Refer to the Appendix A.2 for details. There are two memory ownership modes available to the user:

- either the API owns the record buffer memory; or
- this memory is owned by the user.

The mode is configured via the parameter `memory_owner` and defaults to `ADQ_MEMORY_OWNER_API`. In this mode, record buffers are allocated and resized as needed to handle the incoming data rate. However, the memory consumption of this process is bounded for each channel by the parameters `record_buffer_size_max`, `record_buffer_size_increment` and `nof_record_buffers_max`. Together they specify how

large a record buffer is allowed to grow, the size added in each reallocation and the maximum number of buffers. The record buffer memory is freed when `StopDataAcquisition()` is called.

When the user is responsible for allocating the record buffer memory, references to these memory regions must be registered for each active channel by calling `ReturnRecordBuffer()` prior to initiating the data acquisition with `StartDataAcquisition()`. The reallocation mechanism can still be enabled in this case, allowing the user to allocate an initial buffer that grows in size as needed up to the maximum value specified by `record_buffer_size_max`. Set this parameter to zero to disable this behavior.

3.5 Program Flowcharts

The expected program flow differs slightly depending on the memory ownership (Section 3.4). Figs. 3 and 4 present flowcharts for the two memory modes described in Section 3.4. The steps are labeled on the left-hand side in each figure and are explained in the following list.

> **Note**
>
> The program flow in Fig. 3 is implemented in the software example `gen3_streaming`. This example code in C is available as part of the software development kit (SDK).

1. The first step is configuration. Since this user guide only deals with the aspects of data acquisition, transfer and readout, configuration of other supporting functions are assumed to have been carried out beforehand. Refer to the ADQAPI reference guide [1] for information on how the other supporting functions are configured. The order in which parameters are set is *not* important, only that they are set before `StartDataAcquisition()` is called. The recommended configuration sequence is to first call `InitializeParameters()` to seed a parameter set with default values, update the parameters as needed and finally call `SetParameters()` to make them take effect. Be prepared to handle errors from both function calls.

   (a) Set `memory_owner` to `ADQ_MEMORY_OWNER_API` (default). Either use the default settings or modify the parameters bounding the memory consumption: `record_buffer_size_max`, `record_buffer_size_increment` and `nof_record_buffers_max`.

   (b) Set `memory_owner` to `ADQ_MEMORY_OWNER_USER`.

2. If `memory_owner` is set to `ADQ_MEMORY_OWNER_USER`, the user is expected to allocate an appropriate number of record buffers (see Section 3.8) following the `ADQRecord` format and register these with `ReturnRecordBuffer()`.

3. 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. 2) is created and the API *assumes control* of the digitizer. From this point, the user *must not* call any API functions other than those listed in Appendix A.4 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 is running.

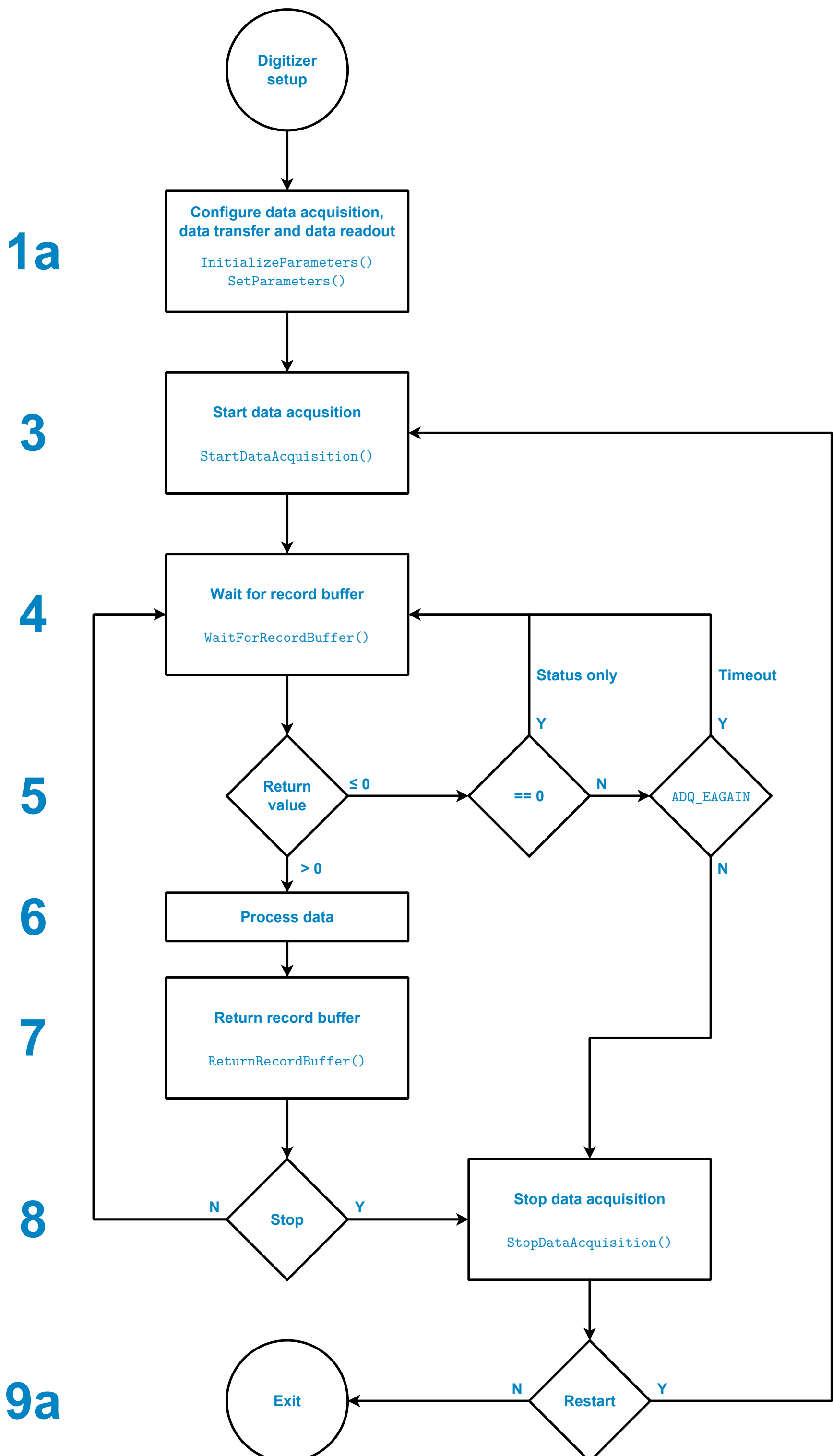

Figure 3: A flowchart for the data readout process with API-owned memory. The steps are labeled on the left-hand side and have a matching entry in Section 3.5.

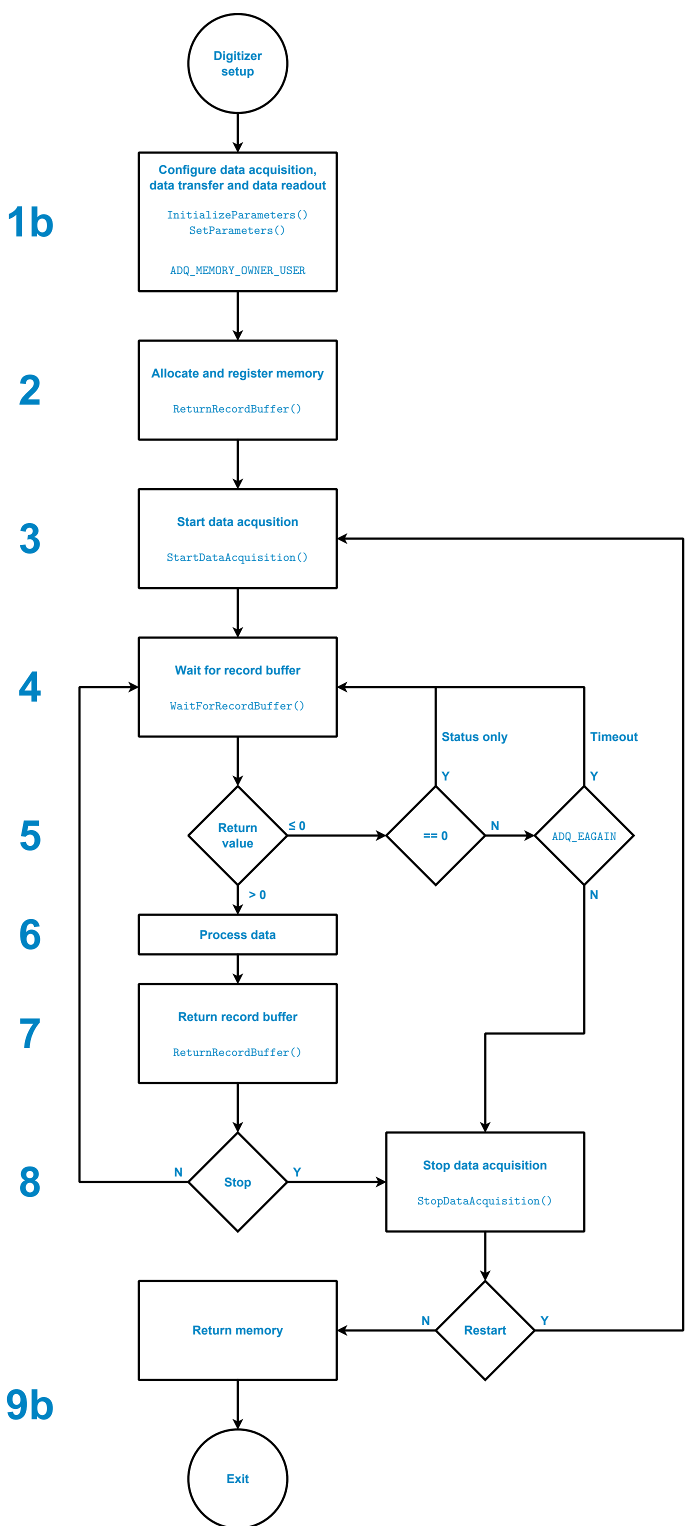

Figure 4: A flowchart for the data readout process with user-owned memory. The steps are labeled on the left-hand side and have a matching entry in Section 3.5.

4. 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.

5. 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. If the return value is `0`, the call was successful but the record should not be accessed (`buffer` will be set to `NULL`). This is a *status event* (Section 3.7) indicating that only the optional parameter `status` can be read. Apart from the error code `ADQ_EAGAIN (-2)` which indicates a timeout, the negative values imply that an unrecoverable error has occurred and that the acquisition has been aborted. The user is expected to call `StopDataAcquisition()` to acknowledge this event (step 8).

6. 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 realtime 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 data readout loop (steps 4 to 8). This affects the balancing of the interface (Section 3.8) and translates directly into requirements on the number of record buffers in rotation.

7. `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*. See Section 3.8 for more information.

8. At the end of the data collection loop, the application should determine if the acquisition is complete or should stop for any other reason. If not, the program flow restarts from step 4. Otherwise, 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 (-5)` may be an expected error code if an unbounded acquisition is stopped.

> **Important**
>
> When `StopDataAcquisition()` returns, any memory owned by the API is returned to the operating system. If the parameter `memory_owner` is set to `ADQ_MEMORY_OWNER_API`, accessing record buffers after this point may lead to access violations.

9. 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 3.

   If the application should exit

   (a) and `memory_owner` is set to `ADQ_MEMORY_OWNER_API`, the application can close immediately; otherwise

   (b) if `memory_owner` is set to `ADQ_MEMORY_OWNER_USER`, make sure to free any dynamically allocated memory from step 2.

3.6 Incomplete Records

The default behavior of the data readout interface is to return complete records, i.e. one record buffer (ADQRecord) holds exactly the data associated with one acquired record. However, there are use cases where this representation is impractical. For example, if the record length is unbounded (ADQ_INFINITE_-RECORD_LENGTH) or just sufficiently large (but finite) there are memory issues associated with attempting to hold the entire record in RAM at the same time. The parameter incomplete_records_enabled exists to tackle this problem. Without this parameter, the overflow mechanism (Section 3.8) would activate and yield records with missing data at the end.

This parameter is set to 0 by default since the intuitive behavior of the record buffer interface is to yield intact records. However, as outlined above, there are situations where this model cannot offer a practical solution. Setting this parameter to 1 increases the flexibility of the interface, at the cost of increased complexity in the logic of the user application.

With incomplete_records_enabled set to 1, WaitForRecordBuffer() now has the possibility to return partial data on success, in addition to its previous behavior (see Section 3.8.2). Whether a record is complete or incomplete is communicated via a flag in the parameter status. Additionally, while the record header is only fully valid for a complete record, there are some values that can be trusted at any time; refer to the ADQRecordHeader documentation for details.

3.7 Status Events

Sometimes there is a need for the API to notify the user about certain events without propagating record data, e.g. signaling that data has been discarded due to a shortage of usable record buffers (see Section 3.8). These *status events* are passed to the user application in the same way as record buffers—via WaitForRecordBuffer(). They may be recognized by looking at the function's return value; a status event is recognized by the return value 0. This indicates that no record data is available for reading, enforced by the fact that buffer is set to NULL. Receiving a status event from WaitForRecordBuffer() should not be matched by a symmetric call to ReturnRecordBuffer() since there is nothing to return.

3.8 Overflow

The streaming interface features a data discarding mechanism to deal with various overflow conditions. 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 data readout interface is consuming record buffers faster than they are being returned.

3.8.1 Physical Interface (case 1)

All digitizers are equipped with on-board memory and while the size varies by model, its function is the same: 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. When data is discarded by the digitizer, there are four possible outcomes when looking at a record transferred under these conditions:

- data is missing *at the beginning* of the record,
- data is missing *from within* the record,
- data is missing *at the end* of the record; or
- the entire record is missing.

These conditions are all discernable from the header member `RecordStatus`. However, the loss of an entire record, or the end of one, is not detectable until pieces of a later record are transferred successfully.

3.8.2 Readout Interface (case 2)

The other critical point is when the transferred data is parsed by the API (Section 3.2) and copied to record buffers (Section 3.4) which propagate to the user application via `WaitForRecordBuffer()`. In this case, there are two possible overflow scenarios (described from the perspective of the internal thread):

- There is no space remaining in the record buffer, i.e. the acquired record is too large. In this case,
  - if incomplete records are allowed (Section 3.6), the partial result is passed to the user application and a new record buffer is retrieved to accommodate the overflowing data;
  - otherwise, the record header member `RecordStatus` is marked to be missing data at the end and the overflowing data is discarded.
- There is no record buffer available. The interface is said to be *starving* and will emit a status event (Section 3.7), notifying the user of this situation. Once this status event has been sent, the thread will suspend all operations until a record buffer becomes available.

When the thread suspends its operations, data stops being transferred over the physical interface. In turn, this causes the digitizer's on-board memory to start filling up and potentially trigger the overflow behavior described in Section 3.8.1. It is worth repeating that no data is lost until the digitizer's on-board buffer memory overflows. Data waiting to be transferred or parsed remains intact when the thread suspends its operations.

To avoid starving the interface, the user first needs to ensure that any processing step (Section 3.5, step 6) 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 number of record buffers rotating in the interface. This number scales with the time the buffer spends in the user application, i.e. the time between `WaitForRecordBuffer()` and `ReturnRecordBuffer()`. Each use case will have its own optimal number of record buffers so this number must be tuned by the user. A structured approach to the balancing is to start with a few buffers, e.g. 32, and subject the readout interface to the expected worst-case data rate. If status events start to appear that indicate that the interface is running out of record buffers, increase the number of record buffers and restart the acquisition. A well-balanced system should not experience this type of status event.

Note

To avoid starving the interface, the user needs to make sure that the processing step (Section 3.5, step 6) is able to handle the acquisition data rate and then balance the number of record buffers in rotation. A well-balanced system should not experience status events indicating that the interface is running out of record buffers.

References

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

A API Reference

This appendix contains the documentation of the enumerations, structures and functions required to compose an application capable of continuously acquiring and transferring data to a host computer. These descriptions are intended to complement the ADQAPI reference guide [1].

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 Enumerations

ADQMemoryOwner 16
ADQParameterId 16
ADQEventSource 17
ADQEdge 17

```
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`.

```
enum ADQParameterId {
    ADQ_PARAMETER_ID_DATA_ACQUISITION = 0,
    ADQ_PARAMETER_ID_DATA_TRANSFER    = 1,
    ADQ_PARAMETER_ID_DATA_READOUT     = 2
}
```

Description

An enumeration of the parameter set identification numbers used by the configuration functions `InitializeParameters()`, `GetParameters()`, `SetParameters()` and `ValidateParameters()`.

Values

`ADQ_PARAMETER_ID_DATA_ACQUISITION (0)`
: The identification number for the data acquisition parameters, defined by `ADQDataAcquisitionParameters`.

`ADQ_PARAMETER_ID_DATA_TRANSFER (1)`
: The identification number for the data transfer parameters, defined by `ADQDataTransferParameters`.

`ADQ_PARAMETER_ID_DATA_READOUT (2)`

The identification number for the data readout parameters, defined by `ADQDataReadoutParameters`.

```
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_GPIO0                     = 25,
    ADQ_EVENT_SOURCE_GPIO1                     = 26,
    ADQ_EVENT_SOURCE_SMA_GPIO                  = 27
}
```

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 2). Not all digitizer models support all the event sources. Refer to the ADQAPI reference guide [1] for more information.

```
enum ADQEdge {
    ADQ_EDGE_FALLING = 0,
    ADQ_EDGE_RISING  = 1,
    ADQ_EDGE_BOTH    = 2
}
```

Description

An enumeration of the edge selection for functions that use an event source. Not all event sources support edge selection, e.g. `ADQ_EVENT_SOURCE_SOFTWARE` only supports `ADQ_EDGE_RISING`.

A.2 Structures

ADQDataAcquisitionParameters . . . 18
ADQDataAcquisitionParametersCommon . . . 19
ADQDataAcquisitionParametersChannel . . . 19
ADQDataTransferParameters . . . 20
ADQDataTransferParametersCommon . . . 21
ADQDataTransferParametersChannel . . . 22
ADQDataReadoutParameters . . . 23
ADQDataReadoutParametersCommon . . . 24
ADQDataReadoutParametersChannel . . . 25
ADQDataReadoutStatus . . . 26
ADQRecord . . . 26

```
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.

Members

`id (enum ADQParameterId)`

The 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 data acquisition parameters for a channel. The struct at index 0 targets the first channel.

`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;
}
```

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` and the valid range is $[-16384, 2^{32} - 1]$. The step size depends on the base sample rate of the digitizer.

`record_length (int64_t)`

The record length in samples. The value `ADQ_INFINITE_RECORD_LENGTH` may be used to indicate an unbounded record. To read out such a record, the data transfer interface has to allow incomplete records to propagate from `WaitForRecordBuffer()` (see Section 3.6). The default value is `0`.

`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).

`trigger_source (enum ADQEventSource)`

An ADQEventSource whose events are used to trigger a record. The default value is ADQ_EVENT_SOURCE_SOFTWARE. Not all event sources listed in Appendix A.1 are supported.

`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)`

Unused, defaults to 0

```
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.

Members

`id (enum ADQParameterId)`

The 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 data transfer parameters for a channel. The struct at index 0 targets the first channel.

`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. The parameters in this struct are *not used* by the digitizer.

Members

`record_buffer_packed_size (int64_t)`

Unused, defaults to `0`.

`metadata_buffer_packed_size (int64_t)`

Unused, defaults to `0`.

`marker_mode (enum ADQMarkerMode)`

Unused, defaults to `ADQ_MARKER_MODE_AUTO`.

`write_lock_enabled (int32_t)`

Unused, defaults to `0`.

`transfer_records_to_host_enabled (int32_t)`

Unused, defaults to `1`.

`packed_buffers_enabled (int32_t)`

Unused, defaults to `0`.

```
struct ADQDataTransferParametersChannel {
    uint64_t          record_buffer_bus_address;
    uint64_t          metadata_buffer_bus_address;
    uint64_t          marker_buffer_bus_address;
    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;
    volatile void *   metadata_buffer;
    volatile void *   marker_buffer;
    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. Apart from `nof_buffers` and `record_buffer_size`, the parameters in this struct are unused.

Members

`record_buffer_bus_address (uint64_t)`

Unused, defaults to `0`.

`metadata_buffer_bus_address (uint64_t)`

Unused, defaults to `0`.

`marker_buffer_bus_address (uint64_t)`

Unused, defaults to `0`.

`nof_buffers (int64_t)`

The number of transfer buffers (Section 3.2) to allocate. The default value is 8.

> **Important**
>
> Gen3 digitizers (ADQ8, ADQ7 and ADQ14) do not have individual transfer buffers per channel. Data transferred from the digitizer is serialized into a single stream. The values at `channel[0]` are used to configure the transfer buffers. For other channels, this parameter is unused.

`record_size (int64_t)`

Unused, defaults to `0`.

`record_buffer_size (int64_t)`

The size of one transfer buffer (Section 3.2) in bytes.

- On ADQ8, this parameter must be a multiple of 9216 B and defaults to 64 · 9216 B.
- On ADQ14 and ADQ7, this parameter must be a multiple of 1024 B and defaults to 1024 · 1024 B.

The same important information that applies to `nof_buffers` also applies to this parameter.

`metadata_buffer_size (int64_t)`

Unused, defaults to `0`.

`record_buffer_packed_offset (int64_t)`

Unused, defaults to `0`.

`metadata_buffer_packed_offset (int64_t)`

Unused, defaults to `0`.

`record_buffer (volatile void *)`

Unused, defaults to `NULL`.

`metadata_buffer (volatile void *)`

Unused, defaults to `NULL`.

`marker_buffer (volatile void *)`

Unused, defaults to `NULL`.

`record_length_infinite_enabled (int32_t)`

Unused, defaults to `0`.

`metadata_enabled (int32_t)`

Unused, defaults to `1`.

```
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 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 data transfer parameters for a channel. The struct at index 0 targets the first channel.

`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. The digitizer only uses the parameter `memory_owner`. Other parameters do not affect the data transfer process.

Members

`memory_owner (enum ADQMemoryOwner)`

As described in Section 3.4, the memory owner specifies who is responsible for memory management: the API or the user. The following values are allowed:

`ADQ_MEMORY_OWNER_API`

Memory is managed by the API. The user may initiate the data acquisition process at any time and the API will allocate and resize record buffers as needed. However, the memory consumption of this automatic process is bounded for each channel by the parameters `record_buffer_size_max`, `record_buffer_size_increment` and `nof_record_buffers_max`. The flowchart in Fig. 3 presents the expected program logic.

`ADQ_MEMORY_OWNER_USER`

Memory is managed by the user. Before initiating the data acquisition process, the user is expected to register a sufficient amount of record buffers for each active channel using `ReturnRecordBuffer()`. The flowchart in Fig. 4 presents the expected program logic.

The parameter defaults to `ADQ_MEMORY_OWNER_API`.

`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`)
This parameter is only used if `memory_owner` is set to `ADQ_MEMORY_OWNER_API`, in which case it specifies an upper limit for the number of record buffers allocated for the channel. The default value is 32.

`record_buffer_size_max` (`int64_t`)
This parameter specifies an upper limit for how large a record buffer is allowed to grow (in bytes). The default value is 1 MiB.

> **Important**
> This parameter is used regardless of the value of `memory_owner`. It is possible to allow the API to grow the buffer while the user is still responsible for the memory (`ADQ_MEMORY_OWNER_USER`). Set the value to zero to prevent this effect.

`record_buffer_size_increment` (`int64_t`)
This parameter is specifies the amount (in bytes) by which a record buffer grows when a reallocation is required. Additionally, the value of this parameter specifies the initial size of a record buffer when `memory_owner` is set to `ADQ_MEMORY_OWNER_API`. The default value is 64 kiB.

`incomplete_records_enabled` (`int32_t`)
As described in Section 3.6, this parameter determines whether or not incomplete records are allowed to propagate to the user via `WaitForRecordBuffer()`. The default value is `0`.

`bytes_per_sample` (`int32_t`)
The number of bytes per sample is used to compute the value of the header field `RecordLength` (which is expressed in samples).

```
struct ADQDataReadoutStatus {
    uint32_t  flags;
}
```

Description

This struct holds status information about a record buffer and the health of the transfer process for a specific channel. It is the type of the output parameter `status` in the function `WaitForRecordBuffer()`.

Members

`flags (uint32_t)`

This member is a 32-bit wide bit field holding status flags. An “all clear” status is represented by all bits being cleared (set to zero).

`Bit 0` `ADQ_DATA_READOUT_STATUS_FLAGS_STARVING`

The channel is starved for memory. There is not a sufficient amount of buffers in circulation, causing incoming data to be discarded. Missing data from from a record will be indicated by the corresponding status bits in the record header `ADQRecordHeader`.

`Bit 1` `ADQ_DATA_READOUT_STATUS_FLAGS_INCOMPLETE`

The record is incomplete, see Section 3.6 for details.

```
struct ADQRecord {
    struct ADQRecordHeader * 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 ADQRecordHeader *)`

A pointer to an `ADQRecordHeader`.

`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()`. Depending on the value of `memory_owner`, the following must be considered:

`ADQ_MEMORY_MODE_API`

The size is set by the API and should be preserved as-is until the buffer is returned with `ReturnRecordBuffer()`.

`ADQ_MEMORY_MODE_USER`

The size should be set by the user at the time of allocation, before registering the buffer with `ReturnRecordBuffer()`.

```
struct ADQRecordHeader {
    uint8_t   RecordStatus;
    uint8_t   UserID;
    uint8_t   Channel;
    uint8_t   DataFormat;
    uint32_t  SerialNumber;
    uint32_t  RecordNumber;
    int32_t   SamplePeriod;
    uint64_t  Timestamp;
    int64_t   RecordStart;
    uint32_t  RecordLength;
    uint16_t  GeneralPurpose0;
    uint16_t  GeneralPurpose1;
}
```

Header structure contained in an `ADQRecord`.

Members

`RecordStatus` (`uint8_t`)

This member is a 8-bit wide bit field holding status information about the record. Bits 0-3 indicate the results of an overflow condition described in Section 3.8.1.

`Bit 0`

If this bit is set, one or several records have been lost preceding this record. Use the member `RecordNumber` to determine how many.

`Bit 1`

If this bit is set, data is missing at the start of the record.

`Bit 2`

If this bit is set, data is missing from within the record.

`Bit 3`

If this bit is set, data is missing at the end of the record.

`Bits 4-6`

This value holds the *fill factor* of the digitizer's on-board memory. The value is given in eight discrete steps: 0-7 and represent that *at least* $N/8$ of the total capacity of the digitizer's on-board memory was occupied when this record was acquired.

`Bit 7`

The record contains samples that have saturated to either the maximum or the minimum value.

`UserID (uint8_t)`

An 8-bit value that may be set from the development kit.

`Channel (uint8_t)`

The originating channel, zero based.

`DataFormat (uint8_t)`

The binary representation used for the record data:

`0`: 16-bit, 2's complement representation.
`1`: 32-bit, 2's complement representation.

`SerialNumber (uint32_t)`

The decimal part of the digitzer's serial number. For example, this field would be set to `9999` for records acquired by a digitizer with serial number "`SPD-09999`".

`RecordNumber (uint32_t)`

The record number as a 32-bit unsigned value. The first record acquired after `StartDataAcquisition()` will have this field set to zero.

> **Important**
>
> The record number wraps to zero at the maximum value.

`SamplePeriod (int32_t)`

The time between two samples, expressed in units of 25 ps on ADQ8 and ADQ7, and 125 ps on ADQ14.

`Timestamp (uint64_t)`

The timestamp of the trigger event, expressed in units of 25 ps on ADQ8 and ADQ7, and 125 ps on ADQ14.

`RecordStart (int64_t)`

The time between the trigger event and the first sample in the record, expressed in units of 25 ps on ADQ8 and ADQ7, and 125 ps on ADQ14. This means that the timestamp of the first sample in

the record is *the sum* of the values of `Timestamp` and `RecordStart`.

- 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).

`RecordLength (uint32_t)`

The length of the record, expressed in *samples*.

`GeneralPurpose0 (uint16_t)`

Unused, defaults to zero.

`GeneralPurpose1 (uint16_t)`

Unused, defaults to zero.

A.3 Configuration Functions

ValidateStructDefinitions 29
InitializeParameters 30
GetParameters 30
SetParameters 31
ValidateParameters 32

```
int ValidateStructDefinitions(
    int  major,
    int  minor
)
```

Validate the struct definitions used by the user application and the API.

Return value

This function returns `0` if the struct definitions are compatible, `-1` if they are incompatible and `-2` if they are 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_STRUCT_VERSION_MAJOR` and `ADQAPI_STRUCT_VERSION_MINOR`:

```
int result = ADQAPI_ValidateStructDefinitions(ADQAPI_STRUCT_VERSION_MAJOR,
                                              ADQAPI_STRUCT_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_STRUCT_VERSION_MAJOR`.

`minor (int)`

The minor version number. Should always be set to `ADQAPI_STRUCT_VERSION_MINOR`.

```
int InitializeParameters(
    enum ADQParameterId  id,
    void *const          parameters
)
```

Initialize a parameter set to default values.

Return value

If the operation is successful, the return value is set to the size of the initialized parameter set. Negative values are error codes with `ADQ_EINVAL (-1)` as the only possible value.

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 Appendix A.2 for information on the default values for each parameter set.

Parameters

`id (enum ADQParameterId)`

The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL (-1)`. Refer to the enumeration `ADQParameterId` in Appendix A.1 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 (-1)`.

```
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. Negative

values are error codes where the following values are possible:

`ADQ_EINVAL (-1)`
: Invalid input argument. Refer to the section describing the parameters for information on valid values.

`ADQ_EIO (-6)`
: An I/O operation failed. Refer to the trace log for more information.

Description

This function reads the current values of the parameter set `id` into the memory region pointed to by `parameters`.

Parameters

`id (enum ADQParameterId)`
: The parameter set's identification number. Targeting an unsupported parameter set will cause the operation to fail with `ADQ_EINVAL (-1)`. Refer to the enumeration `ADQParameterId` in Appendix A.1 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 (-1)`.

```
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. Negative values are error codes where the following values are possible:

`ADQ_EINVAL (-1)`
: Invalid input argument. Refer to the documentation of the parameter definitions in Appendix A.2 for more information.

`ADQ_EIO (-6)`
: An I/O operation failed. Refer to the trace log for more information.

Description

This function writes the parameter set pointed to by `parameters` to the digitizer.

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 (-1)` .

```
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. Negative values are error codes with `ADQ_EINVAL (-1)` as the only possible value.

Description

This function validates the input parameter set according to the same rules as `SetParameters()`. However, the parameters are *not* applied.

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 (-1)` .

A.4 Data Readout Functions

StartDataAcquisition . . . 33
StopDataAcquisition . . . 33
WaitForRecordBuffer . . . 34
ReturnRecordBuffer . . . 36
FlushDMA . . . 37

```
int StartDataAcquisition(void)
```

Start the data acquisition, data transfer and data readout processes.

Return value

If the operation is successful, `ADQ_EOK (0)` is returned. Negative values are error codes where the following values are possible:

`ADQ_ENOTREADY (-4)`
: The data acquisition process is already running.

`ADQ_EINTERRUPTED (-5)`
: The internal thread encountered an error on startup. Refer to the trace log for more information.

Description

Calling this function starts 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 listed Appendix A.4. Calling `StopDataAcquisition()` returns control to the user.

> **Important**
> Once the data acquisition process has started, the user must not call any API fuctions other than those listed in Appendix A.4.

```
int StopDataAcquisition(void)
```

Stop the data acquisition, data transfer and data readout processes.

Return value

If the operation is successful, `ADQ_EOK (0)` is returned. Additionally, `ADQ_EINTERRUPTED (-5)` may be an expected return value if an unbounded acquisition is stopped. Negative values are error codes where the following values are possible:

`ADQ_EAGAIN (-2)`
: The data acquisition process is not running.

`ADQ_EINTERRUPTED (-5)`
: This may be an expected value when an unbounded acquisition is stopped or if a finite acquisition is ended prematurely.

`ADQ_EIO (-6)`
: An I/O operation failed. Refer to the trace log for more information.

Description

Calling this function stops the data acquisition and data transfer processes in a well-defined manner and returns control of the digitizer to the user. This function *must* be called before disconnecting from the digitizer if an acquisition is running.

> **Important**
>
> If the parameter `memory_owner` is set to `ADQ_MEMORY_OWNER_API`, this function frees the record buffer memory.

```
int64_t WaitForRecordBuffer(
    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 value zero indicates a successful operation, but that only the `status` parameter can be read (see Section 3.7). Negative values are error codes where the following values are possible:

`ADQ_EINVAL (-1)`
: Invalid input argument. Refer to the section describing the parameters for information on valid values.

`ADQ_EAGAIN (-2)`
: The operation timed out.

`ADQ_ENOTREADY (-4)`
: The data acquisition process is not running, i.e. `StartDataAcquisition()` has not been called yet, or was not successful.

`ADQ_EINTERRUPTED (-5)`
: The operation was interrupted. This occurs when the data acquisition process comes to a halt, either from an error, or from a controlled stop from calling `StopDataAcquisition()`.

`ADQ_EEXTERNAL (-7)`

An external error occurred, refer to the trace log for more information.

Description

`WaitForRecordBuffer()` allows access to the read port of a channel. Through this port, a channel can pass record buffers, status information or both. The behavior of the interface changes depending on how the data transfer process is configured (see Section 3.6).

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. 2). 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, that property is determined by the configuration (see Section 3.4).

Though memory ownership may vary, 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 stream 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 not as a 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 (-1)` .

`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 (-1)` .

The buffer points to an `ADQRecord` struct, which is declared in the `ADQAPI.h` header file.

```
/* Declare a pointer to receive the location of a record buffer. */
struct ADQRecord *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
)
```

Register memory to be used by the data transfer process.

Return value

If the operation is successful, `ADQ_EOK (0)` is returned. Negative values are error codes with `ADQ_EINVAL (-1)` as the only possible value. Refer to the section describing the parameters for information on valid values.

Description

`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 allowed if the API owns the memory (see Section 3.4). 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 (-1)` .

The buffer points to an `ADQRecord` struct, which is declared in the `ADQAPI.h` header file. If the API owns the memory used in the data transfer process (see Section 3.4), the value of `buffer` is expected to exactly match the values returned by `WaitForRecordBuffer()`.

```
/* Allocate and register a record buffer (user owned memory). */
#define RECORD_BUFFER_SIZE_MAX 8192

struct ADQRecord record = {0};
record.header = malloc(sizeof(ADQRecordHeader));
record.data = malloc(RECORD_BUFFER_SIZE_MAX);
record.size = RECORD_BUFFER_SIZE_MAX;

if (record.header == NULL || record.data == NULL)
{
  /* Handle allocation error. */
}
else
{
  /* Register the record buffer, target the first channel.  */
  int result = ReturnRecordBuffer(0, &record);
  if (result != ADQ_EOK)
  {
    /* Handle error. */
  }
}
```

`int FlushDMA()`

Force the completion of any partially filled transfer buffer.

Return value

If the operation is successful, `1` is returned. Otherwise, `0` is returned.

Description

Calling this function forces any partially filled transfer buffer to fill up and immediately be parsed by the API.

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.