TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ14-FWATD

User Guide

Author(s): SP Devices
Document no.: 16-1849
Classification: Public
Revision: PA7
Date: 2018-03-02

Contents

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

2 System Description **3**
2.1 API . . . 3
2.1.1 SetTransferBuffers . . . 4
2.1.2 FlushPacketOnRecordStop . . . 4
2.1.3 ATDSetupWFA . . . 4
2.1.4 ATDGetAdjustedRecordLength . . . 4
2.1.5 ATDSetWFAPartitionBoundariesDefault . . . 5
2.1.6 ATDGetWFAPartitionBoundaries . . . 5
2.1.7 ATDSetWFAPartitionBoundaries . . . 5
2.1.8 ATDGetDeviceNofAccumulations . . . 5
2.1.9 ATDRegisterWFABuffer . . . 6
2.1.10 ATDStartWFA . . . 6
2.1.11 ATDWaitForWFABuffer . . . 6
2.1.12 ATDStopWFA . . . 6
2.1.13 ATDWaitForWFACompletion . . . 7
2.1.14 ATDGetWFAStatus . . . 7
2.2 Accumulation Engine . . . 7
2.2.1 Partitioning Algorithm . . . 8
2.2.2 Record Segmentation . . . 9
2.2.3 Repeated Accumulations . . . 9
2.3 Physical Interface . . . 10
2.4 Collection Modes . . . 11
2.4.1 Single Measurement . . . 11
2.4.2 Streaming . . . 11
2.5 Overflow Behavior . . . 12
2.6 Low Latency Streaming . . . 12
2.6.1 System Configuration . . . 14

3 Discussion **15**
3.1 Partitioning Algorithm . . . 15
3.2 Record Length & Data Format . . . 16

4 Future Editions **19**

Document History

Revision	Date	Section	Description	Author
PA1	2016-12-19	-	First draft	ME
PA2	2017-05-11	3.1	Now correctly describing Eq. (7) as the *maximum* value for the upper bound.	ME
PA3	2017-09-22	-	Updated style	ME
PA4	2017-09-29	Fig. 1	Updated figure	ME
PA5	2017-10-26	2.5, Figs. 1, 2, 4	Added section on overflow behavior and data collection flow chart.	ME
PA6	2018-02-06	2.6	Added section describing how to configure low latency data transfers.	ME
PA7	2018-02-16	2.6.1	Minor clarifications	ME

1 Introduction

This documents presents the user guide for the advanced time-domain (ATD) firmware option on ADQ14. The outline of this document is presented below.

- In Section 2, the features of ADQ14-FWATD are discussed. High-level descriptions of the different parts of the waveform averaging engine is presented in order to gain an understanding of the system and develop the basis needed to discuss the limitations.
- In Section 3, the limitations of FWATD are discussed and some motivation as to why specific parameters boundaries exist is provided.
- In Section 4, coming additions to this document is listed.

1.1 Definitions & Abbreviations

Table 2 lists the definitions and abbreviations used in this document and provides an explanation for each entry.

Table 2: Definitions and abbreviations used in this document.

Term	Explanation
AFE	Analog front-end
ATD	Advanced time-domain
API	Application programming interface
Batch	Accumulated record output by the device
FW	Firmware (device FPGA configuration)
PCIe	Peripheral Component Interconnect Express
USB3.0	Universal serial bus (version 3.0)
WFA	Waveform averaging

2 System Description

The advanced time-domain firmware for ADQ14 offers the ability to perform hardware-accelerated averaging of waveforms with safe scaling up to 65536 accumulations. Section 2.1 presents the relevant API functions, definitions retrieved from the ADQAPI reference guide[1].

2.1 API

This section describes the API calls relevant to configuring ADQ14-FWATD and capturing the data. Configuring the analog front-end and the different trigger options is not discussed. The functions are presented with a general description of their purpose. Please refer to the ADQAPI reference guide[1] for parameter descriptions.

2.1.1 SetTransferBuffers

```
int     SetTransferBuffers (
        unsigned int    nof_buffers
        unsigned int    buffer_size
)
```

Returns 1 for success, 0 otherwise.

Sets the number and the size of the data transfer buffers used by the physical interface.

2.1.2 FlushPacketOnRecordStop

```
int     FlushPacketOnRecordStop (
        unsigned int    enable
)
```

Returns 1 for success, 0 otherwise.

Enables or disables the forced flush of a packet on record stop

2.1.3 ATDSetupWFA

```
int     ATDSetupWFA (
        unsigned int    record_length
        unsigned int    nof_pretrig_samples
        unsigned int    nof_holdoff_samples
        unsigned int    nof_accumulations
        unsigned int    nof_repeats
)
```

Returns 1 for success, 0 otherwise.

Sets up the ATD WFA module. Performs factorization of the record length into segments and divides the accumulation work load between the device and the host computer. Also sets up the number of pre-trigger or hold-off samples as well as the number of repeated accumulations.

2.1.4 ATDGetAdjustedRecordLength

```
int     ATDGetAdjustedRecordLength (
        unsigned int    record_length
                 int    search_direction
)
```

Returns The adjusted record length. Negative numbers are error codes.

Performs factorization of the record length and potentially suggests a better value to achieve optimal segment length.

2.1.5 ATDSetWFAPartitionBoundariesDefault

```
int    ATDSetWFAPartitionBoundariesDefault (
       void
)
```

Returns 1 for success, 0 otherwise.

Restore the default boundaries for the partitioning algorithm dividing the accumulation work load.

2.1.6 ATDGetWFAPartitionBoundaries

```
int    ATDGetWFAPartitionBoundaries (
       unsigned int    *partition_lower_bound
       unsigned int    *partition_upper_bound
)
```

Returns 1 for success, 0 otherwise.

Return the current boundaries for the partitioning algorithm dividing the accumulation work load.

2.1.7 ATDSetWFAPartitionBoundaries

```
int    ATDSetWFAPartitionBoundaries (
       unsigned int    partition_lower_bound
       unsigned int    partition_upper_bound
)
```

Returns 1 for success, 0 otherwise.

Set the current boundaries for the partitioning algorithm dividing the accumulation work load. Please read sections 2.2.1 and 3.1 before changing these values.

2.1.8 ATDGetDeviceNofAccumulations

```
unsigned int    ATDGetDeviceNofAccumulations (
       unsigned int    nof_accumulations
)
```

Returns The number of averages performed by the device.

Test the work load partitioning for the current boundary settings.

2.1.9 ATDRegisterWFABuffer

```
int    ATDRegisterWFABuffer (
       unsigned int   channel
                int   *buffer
)
```

Returns 1 for success, 0 otherwise.

Register an empty target buffer in the queue.

2.1.10 ATDStartWFA

```
int    ATDStartWFA (
                 int   **target_buffer
       unsigned char   channels_mask
       unsigned char   blocking
)
```

Returns 1 for success, 0 otherwise.

Start the ATD WFA.

2.1.11 ATDWaitForWFABuffer

```
int    ATDWaitForWFABuffer (
       unsigned int   channel
                int   **buffer
                int   timeout
)
```

Returns 1 for success, 0 otherwise.

Wait for the ATD WFA buffer.

2.1.12 ATDStopWFA

```
int    ATDStopWFA (
       void
)
```

Returns 1 for success, 0 otherwise.

Stop the ATD WFA. **ATDWaitForWFACompletion()** should always be called after calling **ATD-StopWFA()** in order to correctly close the data collection thread.

2.1.13 ATDWaitForWFACompletion

int **ATDWaitForWFACompletion** (
    void
)

Returns 1 for success, 0 otherwise.

Wait until the internal data collection thread has been correctly terminated and the device is ready to close or the WFA to be restarted.

2.1.14 ATDGetWFAStatus

int **ATDGetWFAStatus** (
    unsigned int ***wfa_progress_percent***
    unsigned int ***records_collected***
    unsigned int ***stream_status***
    unsigned int ***wfa_status***
)

Returns 1 for success, 0 otherwise.

Read out the current status of the WFA.

2.2 Accumulation Engine

This section aims to describe the accumulation engine from a user-perspective. Figure 1 presents a high-level block diagram of the entire system—from input signal to user-space buffers. The diagram has been significantly simplified so that the standard functions of the ADQ14 have been bundled together in the blocks labeled 'Acquisition Engine' and 'Transfer Engine'.

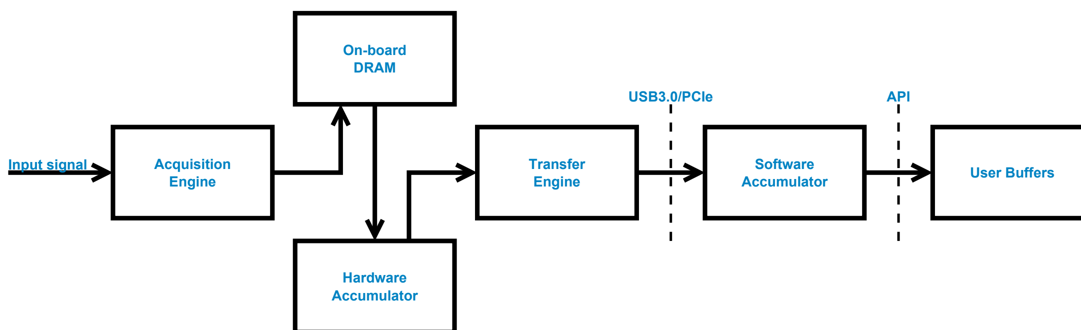


Figure 1: Block diagram of the FWATD averaging system.

The acquisition engine creates records from the input signal which are stored in the on-board DRAM (2 GB capacity). The averaging engine is divided into two parts: one in hardware, and one in software. Thus, the work load is divided between the device and the host computer. This split is determined by a partitioning algorithm applied to the parameter *nof_accumulations* in **ATDSetupWFA()**. The goal of the

algorithm is to determine the most suitable factorization of the number of accumulations that the user wishes to perform (refer to Section 2.2.1 for a detailed description of the algorithm).

We define a *batch* as the resulting record output by the device after performing its assigned number of accumulations. The batch is transferred over the physical interface (ideally USB3.0 or PCIe) to the host computer where a collection thread is performing the last accumulation step by combining a sequence of batches into one record which is delivered to the user buffers.

Since the on-board DRAM has been re-tasked with realizing part of the WFA algorithm, the device-to-host link is much more sensitive to stalls by the host computer as the only buffers available are small (compared to the DRAM) due to residing in the FPGA block RAM.

> **Note**
>
> In ADQ14-FWATD, the DRAM no longer functions as a large FIFO able to buffer large amounts of data in the event that the host computer is busy.

2.2.1 Partitioning Algorithm

Suppose the number of accumulations, N_A, is factorized into

$$N_A = N_{ADQ} \cdot N_{PC} - R, \tag{1}$$

where N_{ADQ} and N_{PC} is the number of averaging steps performed by the device and host computer, respectively. The remainder R is required to allow N_A to assume all integer values in the supported range. It follows that the product $N_{ADQ} \cdot N_{PC}$ will always be equal to or larger than the user- specified value. Following the earlier definition, we can identify a batch as a record averaged N_{ADQ} times and N_{PC} as the number of batches. The device handles the remainder R by only performing N_{ADQ} - R averages in the final batch.

> **Note**
>
> The factorization result of *nof_accumulations* is output in the trace log file.

The goal of the partitioning algorithm is to select an optimal integer value for N_{ADQ} constrained by the boundaries N_L and N_U such that

$$N_{ADQ} \in [N_L, N_U]. \tag{2}$$

These boundaries can be set and read by the user through the functions **ATDSetWFAPartitionBoundaries()** and **ATDGetWFAPartitionBoundaries()**, respectively. However, performing changes to these boundaries may cause the system to no longer fulfill the standard specification[2] with regards to the maximum record length and/or duty cycle. The default boundary values can be restored by calling **ATDSetWFAPartitionBoundariesDefault()**. Please refer to Section 3.1 for an in-depth discussion on the effects of changing these parameters.

> **Note**
>
> Altering the factorization boundaries will cause the WFA performance to change with regards to maximum supported record length and/or duty cycle. Use with caution.

Additionally, this partitioned approach does not result in a data rate reduction by a factor of N_A over the physical interface, but rather the factor $N_{ADQ}/2$, as determined by the partitioning algorithm.

Note

The raw data rate is not reduced by a factor equal to the number of accumulations over the device-to-host interface.

2.2.2 Record Segmentation

In order to manage accumulation of very large records, the record is split into segments of more manageable size. Due to reasons explained in Section 3.2, the record is required to be a multiple of a certain number of samples. Define this number as the *length factor*.

Table 3: Record length factor for all ADQ14 variants.

Device	Length factor
ADQ14-2A/-4A	36
ADQ14-2C/-4C	72
ADQ14-2X/-1X	144

How the record is split into these segments is also determined by a partitioning algorithm, this time applied to the *record_length* parameter in **ATDSetupWFA()**. Due to reasons which fall outside the scope of this document, certain values of the record length will work better than others in terms of allowing a higher duty cycle. In these cases, the partitioning algorithm is able to find an efficient segmentation of the record, allowing the hardware accumulator to work with minimal overhead. These optimal values are in no way intuitive which is why the function **ATDGetAdjustedRecordLength()** exists. This function uses the input record length and a search direction to return an adjusted record length which is able to fulfill the duty cycle specification[2]. If this function is used, it is important to allocate the user-space buffers using this adjusted value. Otherwise, segmentation faults are likely to occur.

Note

The function **ATDGetAdjustedRecordLength()** should be used to find a record length which is optimal from a hardware perspective. Any allocation of user-space target buffers should use the adjusted value to avoid segmentation faults.

2.2.3 Repeated Accumulations

A common wish is to be able to perform multiple waveform accumulations back-to- back, i.e., the first N_A triggers resulting in one record in the user-space, the next N_A triggers generating a second record, and so on. The parameter *nof_repeats*, N_R, in **ATDSetupWFA()** exists for this purpose. The accumulation engine uses this parameter to determine the number of user-space records to produce. The accumulation process will continue until the completion criteria is met, i.e. the hardware has captured $N_R \cdot N_A$ triggers and they have all been transferred to the host computer, or the user aborts the process by calling **ATDStopWFA()**[1].

[1] A call to **ATDStopWFA()** must always be followed by a call to **ATDWaitForWFACompletion()** in order to properly close the thread spawned by the API.

Additionally, if there is a wish to generate a constant stream of averaged records, the number of repeats should be set to 2^{32} or `UINT_MAX`. This mode requires the user to implement the *streaming* data collection mode detailed in Section 2.4.2 and will require calls to **ATDStopWFA()** followed by **ATDWaitForWFACompletion()** in order to correctly end the collection loop.

Note

The number of repeats determines how many user-space records are generated. A value of 2^{31} - 1 enables infinite streaming, requiring the user to implement the *streaming* data collection mode (see Section 2.4.2).

2.3 Physical Interface

Transferring data from the device to the host computer is done over a physical interface. ADQ14 supports USB3.0 (USB2.0[2] fallback), PXIe, PCIe and μTCA, although not every variant of ADQ14 supports every interface. For example, an ADQ14-PXIe device cannot be used as a PCIe device and an ADQ14-USB cannot be used as a PXIe device with regard to how it is connected to the host PC.

Note

PXIe and μTCA devices use the PCIe protocol to transfer data.

These interfaces have different specifications on the maximum transfer rate with USB3.0 being the slowest at a theoretical limit[3]of roughly 500 MB/s (effective data rate). However, achieving these transfer rates is dependent on how the user sets up the API transfer buffers. In the USB case, failure to allocate enough memory will cause significant overhead, causing the effective transfer rate to drop. PCIe is more robust to the transfer buffer size.

The transfer buffers should be set up in the user code by calling **SetTransferBuffers()**. This function allows the user to specify the number of transfer buffers to allocate in parallel, and their size. Once a buffer is filled, the API will parse the contents, rendering records which are later presented to the user. While this process is ongoing, the buffer is marked as full and no new transactions on the physical interface can target this buffer. In order not to block the flow of data, there is a need for several buffers in parallel. Naturally, the minimum requirement is two buffers but allocating a few more is recommended.

Additionally, large buffers result in a device-to-host interface able to reach higher transfer speeds. The allocated size is critical to achieving high transfer rates over USB, less so for PCIe. Transfer buffers in the range 1 MB – 4 MB is recommended. The API will attempt to allocate memory linearly for each transfer buffer. Should this operation fail, the call to **SetTransferBuffers()** will return 0 and a message will be visible in the trace log file.

Note

Allocating several transfer buffers in parallel is needed in order not to block the flow of data over the physical interface. The recommended size is in the range 1 MB – 4 MB. Memory is allocated linearly.

[2]USB2.0 is limited to a transfer rate of roughly 57 MB/s (480 Mbps), ideally. SP Devices strongly advises against using your digitizer configured for FWATD with this interface. Refer to the discussion in Section 3.1 for more details.

[3]This speed is not guaranteed by ADQ14-USB devices.

2.4 Collection Modes

This section presents an overview of the two data collection modes available to the user. The streaming mode (Section 2.4.2) is more general than the single measurement mode (Section 2.4.1) and is the recommended mode of operation.

2.4.1 Single Measurement

The single measurement mode is well suited to observing short events and is simple with respect to the required user code. The mode is activated by specifying an array of valid target buffer pointers to **ATDStartWFA()**. The available memory pointed to must be at least

$$4N_R L_R \quad \text{bytes},$$

where N_R is the number of repeats and L_R is the record length in samples. The channel mask will determine which channels are active and thus which array entries are valid pointers.

Since it is required to allocate all the memory needed for the measurement at once, this mode is not well suited for long records combined with a high number of repeats, instead the streaming mode (Section 2.4.2) should be used.

Note

The FWATD Python 3 example demonstrates how the single measurement mode is implemented.

2.4.2 Streaming

The streaming mode is based around a buffer queue system controlled by the API functions **ATDRegisterWFABuffer()** and **ATDWaitForWFABuffer()**. The mode is activated by specifying `NULL` as the value of the target buffer array in **ATDStartWFA()**.

Each buffer represent a record and before calling **ATDStartWFA()** a sufficiently large number of buffers should be registered for each channel intended to be used. Completed records are delivered to the user via the function **ATDWaitForWFABuffer()** by reporting a location in the memory (same as the previously registered area) where the latest completed record can be read. The user is responsible for memory allocated for the buffer queue and the queue is reset in the call to **ATDStopWFA()**.

Note

The user is responsible for the allocated memory at all times. The clearing of the queue in **ATDStopWFA()** does not free the memory.

The minimum number of buffers needed to be registered will depend on multiple factors such as the transfer buffer size, the data throughput and the number of repeats, i.e. varying with the use-case. The goal is to keep the system balanced so that there always is a buffer available to place the accumulated record. In short, the queue must never be empty while the measurement is ongoing. Since large transfer buffers are recommended (see Section 2.3), the parsing of this buffer may generate several user-space records at once, thus requiring the queue of empty target buffers to be filled to a certain threshold. Generally, a large number of target buffers (≥ 50) is recommended. However, if the memory can be afforded, an even larger number may have additional benefits.

> **Note**
>
> The queue system has to be balanced such that empty target buffers are available whenever the need arises.

Once the user has processed the accumulated record, e.g. by writing the data to disk, the target buffer should be recycled as an empty buffer by calling **ATDRegisterWFABuffer()**.

> **Note**
>
> The FWATD C example demonstrates how the streaming mode is implemented.

Data Collection Flowchart

Figure 2 presents a flowchart of the data collection loop.

2.5 Overflow Behavior

In this section, the term 'overflow' refers to data being discarded due to a bandwidth mismatch in the physical interface or due to a violation of the maximum supported duty cycle.

ADQ14-FWATD does not keep track of where data has been discarded in the stream of records, only that this has occurred. The user us notified of this event through the WFA status, read by calling **ATDGetWFAStatus()**. The status is *sticky*, meaning it reflects all the status codes that have occurred during the measurement. Configuring the WFA for a new measurement (**ATDSetupWFA()**) clears the status value.

In the event of an overflow, it is imperative that the WFA engine is shut down in a well-defined manner in order to ensure correct behavior once it is restarted. A few operations are required from a user perspective: detecting the overflow condition and closing the internal streaming thread.

For a minimal flow control, no polling mechanism is required. Instead, the status is indirectly provided through the return value of **ATDWaitForWFABuffer()** together with the address of the returned buffer. If an overflow is detected by the internal streaming thread, the buffer queue is placed into a state in which subsequent calls to **ATDWaitForWFABuffer()** will return the value '0' with the pointer value '-1'.

2.6 Low Latency Streaming

Some applications may require that the data buffer arrives in user space as soon as possible after its corresponding raw data has been captured. By default, ADQ14-FWATD premiers high throughput by recommending large transfer buffers and allowing the data to drive the output rate.

While setting up smaller transfer buffers would reduce the latency, the action on its own is not enough to solve the problem and still keep the system robust. Since the data drives the output (low overhead), there are no guarantees that the data representing one user space record constitutes an even number of transfer buffers. This mismatch creates suboptimal transfer performance when combined with small buffer sizes.

For each use case, the solution to the problem involves

- minimizing the required bandwidth of the device-to-host interface,
- activating an internal flush mechanism and

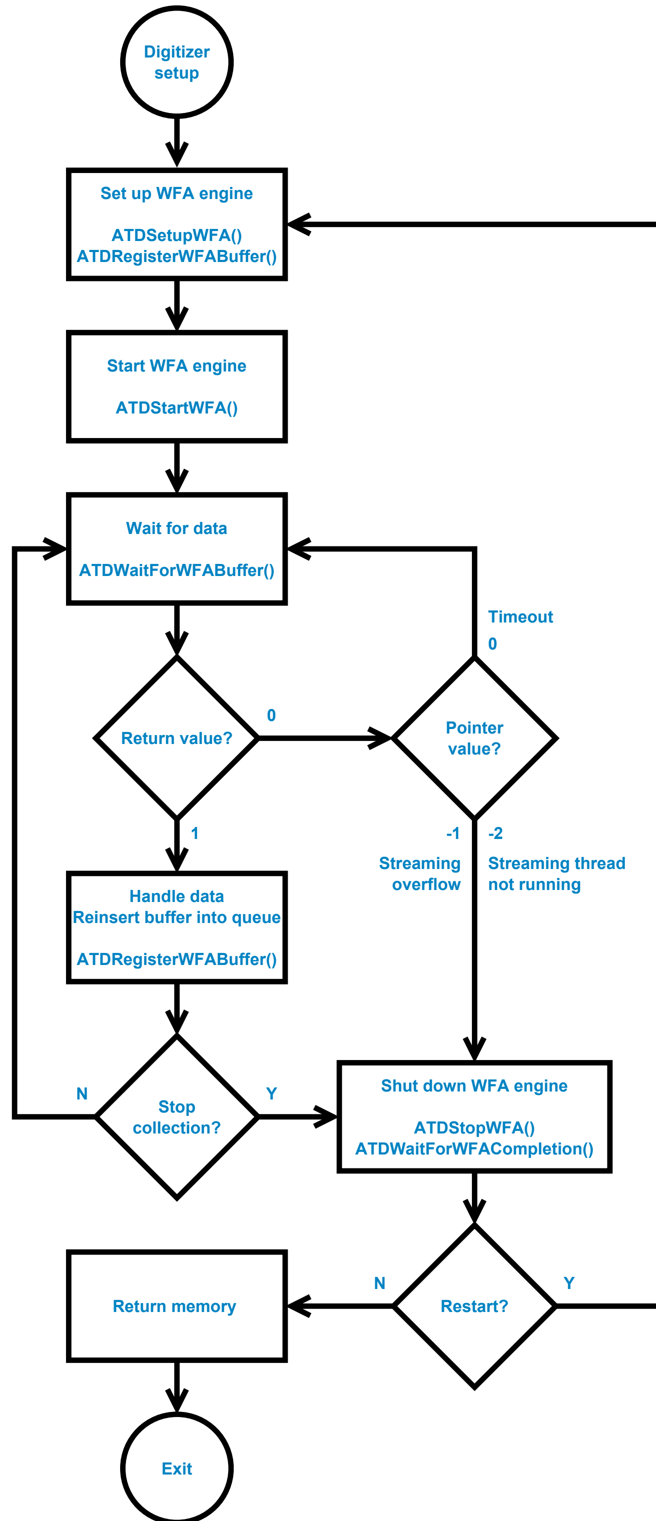


Figure 2: Data collection flowchart for the streaming mode.

- computing and tuning the optimal transfer buffer size.

The internal flush mechanism intentionally adds overhead in order to ensure that every user space record consists of a fixed number of *packets*. Packets represent abstracted data and are what constitutes the data in a transfer buffer.

Note

Activating the internal flush mechanism requires ADQAPI r35550 or later.

The resulting system will be responsive and perform well for the attuned use case, but will be less robust to other stimuli patterns.

2.6.1 System Configuration

The following steps outline the process of configuring low latency data acquisition for a particular use case. A *use case* is characterized by its requirements on the record length, R_L, the number of accumulations N_A, the trigger period T_P, the maximum tolerable latency $T_{\text{lat, max}}$ and the device sample rate. The record length R_L is assumed to fulfill the requirements outlined in Sections 2.2.2 and 3.2.

1. Optimize the WFA partition boundaries to relax the requirements on the device-to-host interface. Increase the upper bound to

$$N_U = \left\lfloor \frac{1024^3}{\frac{8}{9} \cdot 2 \cdot 4 \cdot R_L} \right\rfloor$$

   by calling **ATDSetWFAPartitionBoundaries()**. Refer to Sections 2.2.1 and 3.1 for details on this equation.

2. Compute the required transfer buffer size to fit exactly one record. Equation (3) give the size in bytes for a –C device (1 GSPS base sample rate) and Eq. (4) give the size in bytes for an –X device (2 GSPS base sample rate).

$$TB = 1024 \left(1 + \left\lceil \frac{R_L \cdot 4 - 1000}{1016} \right\rceil \right) \quad (3)$$

$$TB = 1024 \left(1 + \left\lceil \frac{R_L \cdot 4 - 992}{1008} \right\rceil \right) \quad (4)$$

3. Activate the internal flush mechanism by calling **FlushPacketOnRecordStop()** (requires ADQAPI r35550 or later).

4. (OPTIONAL) If the device-to-host bandwidth is insufficient, the transfer buffer size may be increased to a multiple of the unit size computed in step 2. Let us define the integer k as this factor. The worst-case latency defines an upper bound on its value as

$$0 < k < \frac{T_{\text{lat, max}}}{T_P \cdot N_A}. \quad (5)$$

The user has some freedom in choosing the value of k, unless the upper bound is less than 1, in which case this method is not a valid option. Depending on the record size, a lower k may be desired to keep the transfer buffer size below e.g. 4 MB.

This will increase the bandwidth of the device-to-host interface but cause records to have non-uniform (but predictable) latency. For example, for a transfer buffer that fits precisely three user space record, the perceived latency of the first record will be $2T_PN_A$, the second T_PN_A and the last one effectively zero.

3 Discussion

This section aims to provide a discussion on the different aspects of ADQ14-FWATD needed to be considered before and during integration of this product into any system. A discussion on the effects of the partitioning algorithm can be found in Section 3.1, followed by a discussion on the record length and data format in Section 3.2.

3.1 Partitioning Algorithm

Section 2.2.1 described the purpose of the partitioning algorithm used to determine the work load distribution between the hardware and software accumulator. The algorithm determines values for N_{ADQ}, N_{PC} and R (indirectly) from a given number of accumulations, N_A, according to (1). Assuming the maximum raw data volume from the ADQ14-FWATD, i.e. $\approx$ 14.9 GB/s, it is possible to generate the heat maps presented in Fig. 3 for a few values of the partition boundaries N_L and N_U. This figure presents the maximum achievable duty cycle as a function of the host interface throughput and the user-specified number of accumulations.

Another interesting metric is the DRAM utilization efficiency for the last batch due to the remainder R. Let the efficiency η be defined as the relative size between the last batch and the other batches in terms of the number of raw records (triggers) needed, i.e.

$$\eta = \frac{N_{ADQ} - R}{N_{ADQ}}. \tag{6}$$

Figure 4 presents the corresponding utilization histograms, complementing the data in Fig. 3.

Analyzing the data in Fig. 3, it is evident that tasking the digitizer with performing additional accumulations before transferring the batch alleviates the requirements on the average transfer speed of the host interface. However, if the maximum supported record size is to be maintained, the upper partition boundary must remain. This results in a shrinking set of valid integer values for N_{ADQ}, causing the progressively deteriorating distributions shown in Fig. 4. For low utilization efficiencies, the hardware accumulator will be the bottleneck, potentially causing an overflow due to the large overhead in the memory operations. Therefore, there is a trade-off when only the lower bound is increased where the critical point is moved from the host interface to the hardware accumulator.

Increasing the upper boundary will prevent the set from shrinking and thus avoiding the effects shown in Fig. 4. However, doing so will sacrifice the maximum supported record size. If the use-case is well defined and a record size of 1 MS is not needed, increasing the upper partition boundary may have a

beneficial effect on the performance. To compute the maximum value for the upper bound given a certain record length, refer to Eq. (7)

$$N_U = \left\lfloor \frac{1024^3}{\frac{8}{9} \cdot 2 \cdot 4 \cdot R_L} \right\rfloor, \tag{7}$$

where R_L is the record length.

The lower bound is determined by the minimum required data rate reduction in order to support the physical interface with the lowest average transfer speed. For ADQ14, this is USB3.0 for which an average throughput[4] of 310 MB/s yields

$$N_L = \left\lceil \frac{16 \cdot 10^9}{310 \cdot 1024^2} \right\rceil = 50. \tag{8}$$

Furthermore, it is clear from Fig. 3 that using ADQ14-FWATD with USB2.0 will not be able to reach high averaging duty cycles unless the device is tasked with performing at least 509 accumulations, assuming an average transfer rate of 30 MB/s. This entails that the upper bound must shift upwards as well, limiting the maximum record size to a few hundred kS.

3.2 Record Length & Data Format

Due to bandwidth requirements, the raw record samples are packed and stored as 14-bit values in the on-board DRAM. This is a difference to the other firmware options on ADQ14 where the samples are 16-bit aligned. This packing translates to a compression factor of $\frac{8}{9}$. It is due to this compression that the records have to be a multiple of the length factor, detailed in Tab. 3.

The accumulator uses 32-bit samples in order to guarantee safe scaling for up to 65536 accumulations. This means that the minimum ADC code, -2^{15}, can be accumulated 65536 times without an arithmetic overflow. The same is true for the maximum ADC code, $2^{15} - 1$. The data type is converted in the hardware accumulator which reads out the 14-bit samples from DRAM and accumulates each sample using a 32-bit accumulator. Therefore, ADQ14-FWATD uses 4 bytes per sample when the data is transferred to the host computer.

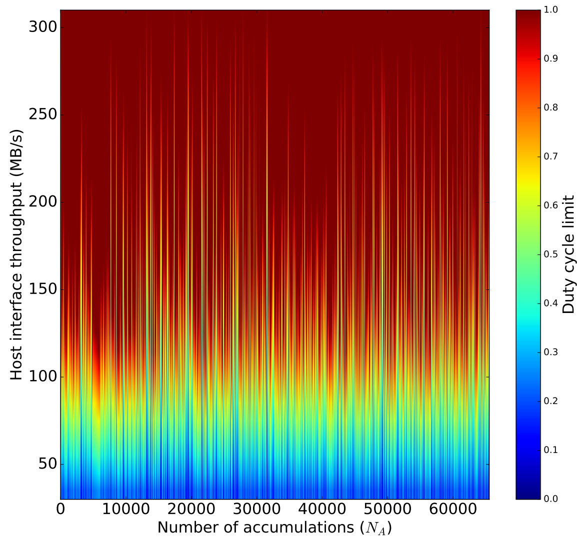
If the target signal does not utilize the full input range of the AFE, the upper bound for the number of accumulations where safe scaling is guaranteed increases.

Note

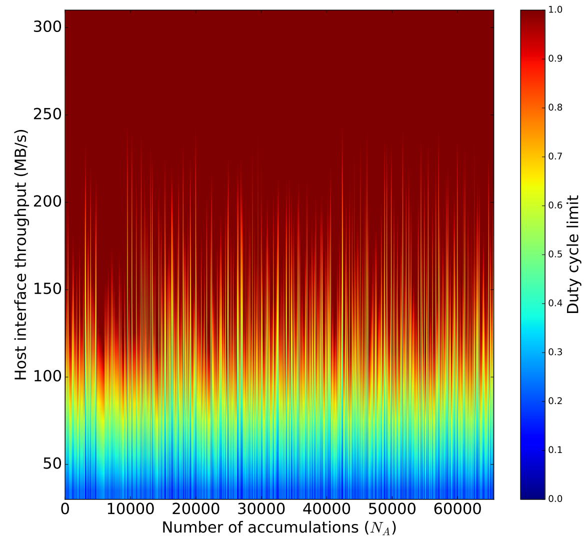
An accumulation overflow is reported by the corresponding bit in the *wfa_status* parameter after calling **ATDGetWFAStatus()**.

[4] The speed of 310 MB/s is empirically determined from system tests.

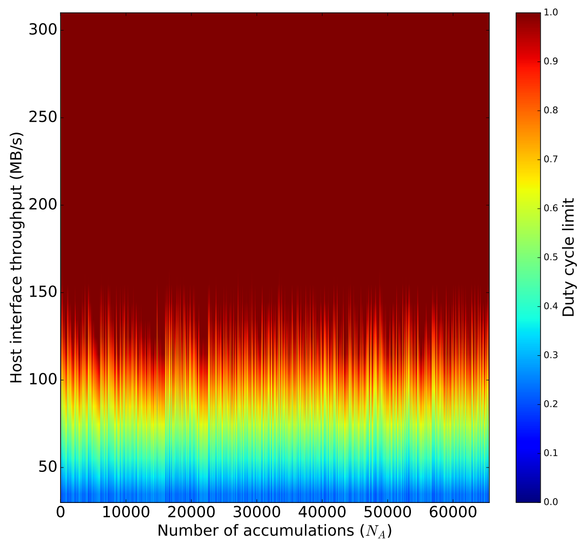
[5] A duty cycle of 100% is not possible in ADQ14-FWATD. Following the end of a record, a re-arm time of 20 ns is required before a new record can begin.



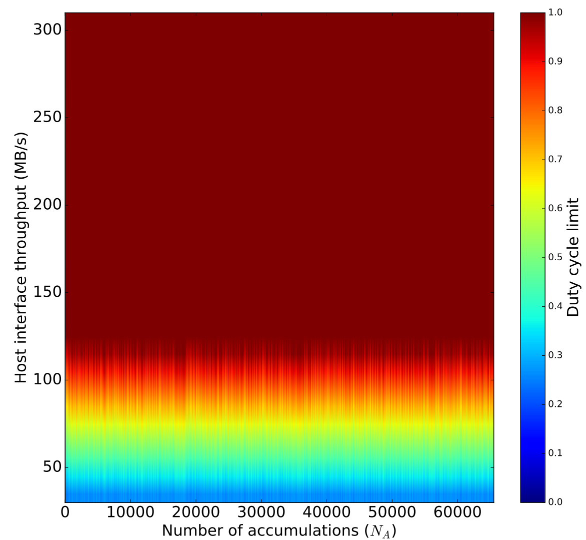
(a) N_L = 50, N_U = 144.



(b) N_L = 64, N_U = 144.

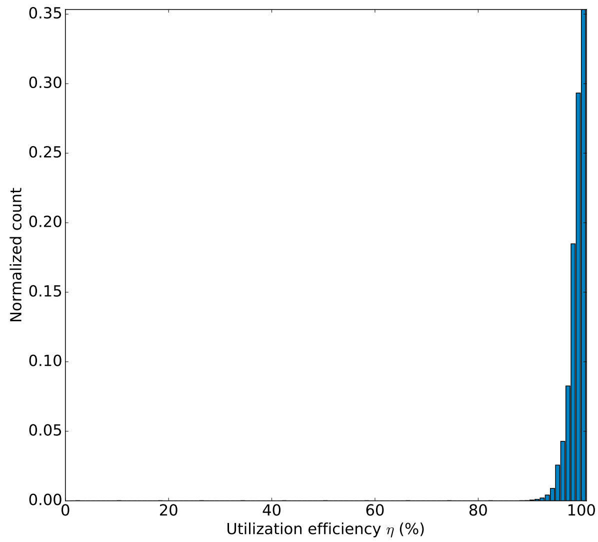


(c) N_L = 100, N_U = 144.

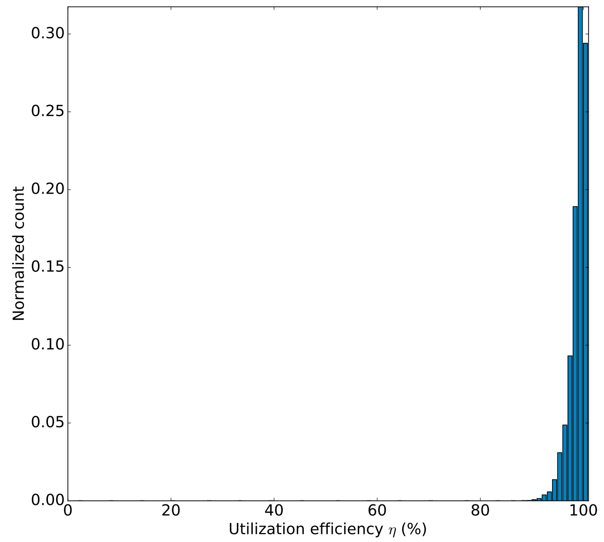


(d) N_L = 128, N_U = 144.

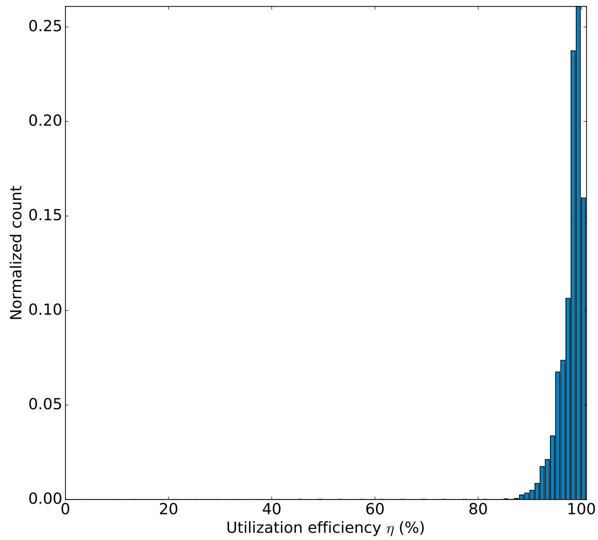
Figure 3: Illustration of maximum duty cycle[5] as a function of the host interface throughput and the user-specified number of accumulations factorized using different values for the partition boundaries.



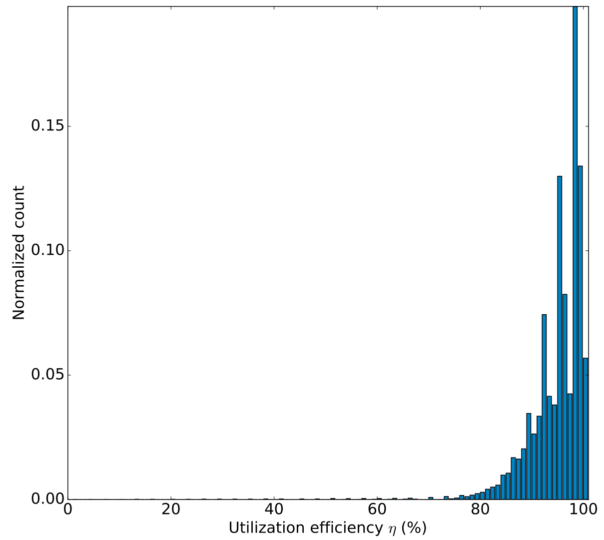
(a) N_L = 50, N_U = 144.



(b) N_L = 64, N_U = 144.



(c) N_L = 100, N_U = 144.



(d) N_L = 128, N_U = 144.

Figure 4: Illustration of DRAM utilization efficiency with a few different values on the partition boundaries.

4 Future Editions

Future editions of this document will include

- a section describing the use of the thresholding filter

References

[1] Signal Processing Devices Sweden AB, *14-1351 ADQAPI Reference Guide*, Dec. 2016. Technical Manual.

[2] Signal Processing Devices Sweden AB, *14-1397 PA8 ADQ14-FWATD Datasheet*, Nov. 2015. Technical Manual.