TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ3 Series FWDAQ Development Kit

User Guide

Author(s): Teledyne SP Devices
Document ID: 20-2507
Classification: Public
Revision: A
Print date: 2021-06-21

Contents

1 Introduction **4**
1.1 Definitions and Abbreviations . . . 4

2 Prerequisites **5**

3 Development Environment and Tools **6**
3.1 Unpacking the Development Kit . . . 6
3.2 Opening the Development Kit . . . 7
3.3 Setting Up the Project . . . 7
3.3.1 Configuration . . . 7
3.4 Building the Design . . . 8
3.5 Working with the Design . . . 8
3.5.1 Typical Design Flow . . . 8
3.6 Analyzing the Implemented Design . . . 9

4 General Concepts **13**
4.1 Parallel Digital Design . . . 13
4.2 Data Flow . . . 13
4.3 Clock Domain Crossing Synchronization . . . 15
4.3.1 CDC Synchronization of a 1-bit Signal . . . 16
4.3.2 CDC Synchronization of a Multi-Bit Signal . . . 17
4.4 AXI Control Bus . . . 18
4.4.1 Control Bus Signals . . . 18
4.5 Data Bus . . . 18
4.5.1 Two Bus Definitions . . . 18

5 Data Bus Signals **20**
5.1 Timestamp . . . 21
5.2 Timestamp Synchronization . . . 21
5.2.1 Sample Index . . . 21
5.2.2 Count . . . 21
5.2.3 Event . . . 22
5.3 Trigger . . . 22
5.3.1 Sample Index . . . 22
5.3.2 Sample Index Fraction . . . 22
5.3.3 Rising . . . 22
5.3.4 Event . . . 22
5.3.5 Inhibit . . . 22
5.4 Overrange . . . 23
5.5 General Purpose . . . 23
5.6 Sample Data . . . 23
5.7 Valid . . . 23
5.8 Record . . . 23

5.8.1 Start, Start Index, Stop and Stop Index . . . 23
5.8.2 Number . . . 25
5.9 User ID . . . 25
5.10 Differences relative to ADQ14, ADQ7 and ADQ8 . . . 25

6 User Logic 1 26
6.1 Default Contents . . . 26
6.2 Register File . . . 26
6.3 Default Register File . . . 27

7 User Logic 2 29
7.1 Default Contents . . . 29
7.2 Register File . . . 29
7.3 Default Register File . . . 30

8 Timing Closure 32

9 Troubleshooting 33
9.1 Debugging on Hardware . . . 33
9.1.1 Creating the Debug Core . . . 33
9.1.2 Connecting to the Debug Core . . . 34

Document History

Revision	Date	Section	Description	Author
A	2021-06-21	-	Initial revision	TSPD

1 Introduction

This document is the user guide for the development kit of the data acquisition firmware for ADQ3 series digitizers. There are different versions of the development kit depending on the device-to-host interface and the target number of channels. Make sure the development kit matches the target hardware.

The development kit centers around the *user logic areas*. These areas target strategic points in the data path and are specifically intended to contain custom HDL designs.

The first user logic area, UL1, described in Section 6, operates on the full-rate data stream—before the trigger information has been decoded to create records. The second user logic area, UL2, described in Section 7, operates on complete records, potentially with a reduced sampling rate.

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
ADC	Analog-to-digital converter
CDC	Clock domain synchronization
Devkit	Development kit
DBT1	Data bus type 1
DBT2	Data bus type 2
FWDAQ	The data acquisition firmware
GiB	Gibibyte (1024^3 bytes)
PCB	Printed circuit board
RTL	Register transfer level
Tcl	Tool command language—scripting language used in Vivado.
UL1	User logic 1—the first open FPGA area, see Section 6.
UL2	User logic 2—the second open FPGA area, see Section 7.
VHDL	VHSIC hardware description language
Verilog	Hardware description language
Vivado	Xilinx FPGA design suite

2 Prerequisites

The development kit has the following prerequisites:

- A license for the development kit purchased from Teledyne SP Devices.
- A license for the Xilinx design tools. For current versions of the development kit, a license for *Vivado 2020.2* is required.
  - Minimum tooling is the Vivado *Design Edition*.
  - The Vivado *WebPack* does *not* support this development kit.
  - Xilinx *ISE* cannot be used.
- Previous experience with defining custom logic using Verilog or VHDL.

3 Development Environment and Tools

This section describes the development kit workflow and the associated tools.

3.1 Unpacking the Development Kit

The development kit is delivered as a `.zip` archive containing the project files, source files and documentation. The first level of the archive contains a `README` file and another archive that is password protected. By unpacking the password protected archive, the user agrees to the terms of the development kit license. Make sure to extract the archive to a directory where the current user has read *and* write permissions.

Important

By unpacking the password protected archive, the user agrees to the terms of the development kit license.

The extracted archive is organized as follows:

```
<Extract root>/
```

`constraints/`	Contains the constraint files for the design.
`documentation/`	Contains the documentation for the development kit.
`framework/`	Contains the files representing the base design along with header files to interact with the data bus. Apart from the file `config.vh` (see Section 3.3.1, the files in this directory should *not* be modified by the user.
`user_logic/`	Contains the Verilog source files for the user logic areas.
`license.txt`	Development kit license file.
`devkit.tcl`	The Tcl file that is sourced to enable the development kit.

Note

Apart from the file `config.vh` (see Section 3.3.1, the files in the `framework` directory are not intended to be edited by the end user.

3.2 Opening the Development Kit

To open the development kit in Vivado, follow the steps outlined below.

1. Start Vivado
2. In the menu bar, select *Tools* > *Run Tcl Script...*
3. Open the file `<Extract root>/devkit.tcl`. The Tcl console will output message similar to this:

```
*** ADQ3 Series Development Kit ***
Revision: r59529
Usage:
  devkit_setup   - Create project
  devkit_build   - Build project
  devkit_analyze - Analyze implemented design
  devkit_mcs     - Generate .mcs firmware file.
                   (Not needed if devkit_build is used.)
```

At this point, the Tcl commands specific to the development kit have been defined and are available in the Tcl console. The project is now ready to be set up for first-time use.

Note

The development kit for ADQ3 series digitizers works differently compared to previous products since building the user logic areas no longer requires special Tcl commands. See Section 3.5 for details.

3.3 Setting Up the Project

To set up the development kit, execute the command

```
devkit_setup
```

in the Tcl console. The process may take a moment to finish. Once the setup is complete, a Vivado project has been created and the design is ready to be built. This step only has to be completed once, after which the following directories will have been created:

```
<Extract root>/
├─ artifacts/
│  ├─ latest/   Contains firmware files from the most recent build.
│  └─ log/      Contains firmware files from previous builds.
├─ build/       Contains all generated project files.
└─ reports/     Contains the results from the development kit analyzer tool (see Section 3.6).
```

3.3.1 Configuration

The file `framework/config.vh` contains parameters that influence the base design. Some parameters should not be changed by the user—these are clearly marked. Other parameters may be modified and

serve as a way to remove modules in the base design to free up FPGA resources, provided that the application does not require the features they provide. For example, the parameter `REMOVE_FILTER` removes the built-in FIR filter if set to a nonzero value. This frees up DSP and routing resources in the FPGA fabric.

3.4 Building the Design

To build the entire design, execute the command

```
devkit_build
```

in the Tcl console. Depending on the computer specifications and the complexity of the design as a whole, i.e. the precomplied design and the user logic together, this may take several hours. Once the process is complete, an `.mcs` file has been generated in the `artifacts/latest/` directory. This file represents a new firmware for the digitizer and may be uploaded using the *ADQUpdater* application. Refer to the ADQUpdater user guide [1] for instructions on how to manage the digitizer's firmware.

3.5 Working with the Design

This section describes the workflow of adding customized logic functions to the digitizer firmware.

3.5.1 Typical Design Flow

This section outlines the typical design flow for the development kit.

1. Set up the development kit project as described in Section 3.3.
2. Modify or insert new Verilog code into `user_logic1.v` or `user_logic2.v`. This operation can be broken down into four steps:
   (a) Extract *data*, *data valid* and other relevant bus signals using the *bus extraction macros* (see Section 5).
   (b) Process the extracted signals, i.e. stimulate the custom user design.
   (c) Insert the processed data, data valid and relevant bus signals using the *bus insertion macros* (see Section 5).
   (d) Set the correct value for the `BUS_PIPELINE` delay parameter to keep the correct time relation between signals that were not manually inserted.
3. Generate the FPGA configuration file (`.mcs` file) by using one of the two methods outlined below:
   - *Automatic*
     (a) Execute the command

     ```
     devkit_build
     ```

     in the Tcl console.
   - *Manual*

(a) Start the build by selecting *Generate Bitstream* in Vivado. This action will rebuild the design and end with the bitstream generation.

(b) Once the bitstream is available, generate the FPGA configuration file by executing the command

```
devkit_mcs
```

in the Tcl console. The configuration (`.mcs`) file can be found in the `artifacts/latest/` directory after the process is complete.

4. Analyze the design for potential problems by executing the command

```
devkit_analyze
```

in the Tcl console and then open the HTML report in the `reports/` folder. See Section 3.6 for details.

5. Program the configuration file representing the custom design into the digitizer using the ADQUpdater application. Refer to the corresponding user guide [1] for details on the programming process.

6. Test the custom firmware using either

   - one of the software examples available in the ADQAPI library or
   - a custom user application.

3.6 Analyzing the Implemented Design

When the implementation step has completed it is highly recommended to analyze the design to detect the most common issues. To do this, execute the command

```
devkit_analyze
```

in the Tcl console. This will run a set of checks and summarize the results into a report. The generated report includes the current results and the difference with respect to the previous report. To keep track of design parameter trends, it is encouraged to generate the report after every run even if no timing errors occurred. The report is saved as an HTML file that should be opened in a web browser, see Fig. 1. There are also several log files produced and placed in the `reports/` folder that offers additional details on the reported issues.

Tip

If a continuous integration tool is used to automate the workflow, consider integrating the output from `devkit_analyze` as a part of the process.

The report is divided into six different sections. Each section has several parameters and result values. For parameters with a maximum value the format is

```
Percentage[Value/Maximum value] (Difference)
```

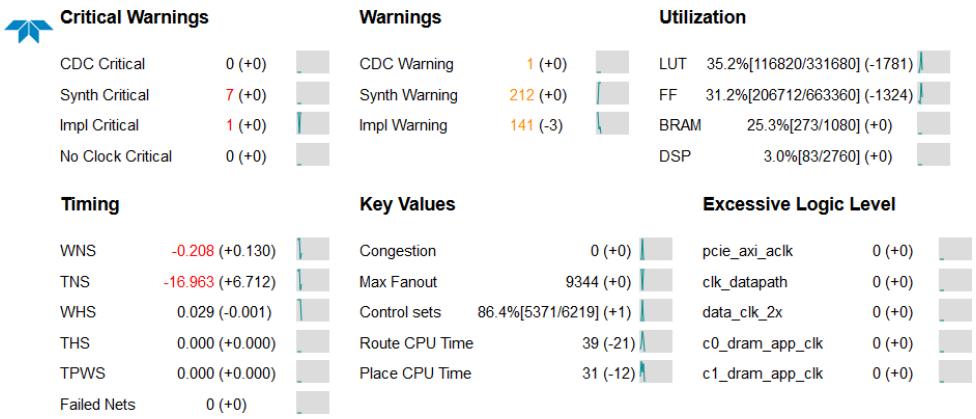

Figure 1: The HTML report from `devkit_analyze` rendered in a web browser.

and values without an upper boundary are formatted as

```
Value (Difference)
```

The difference is calculated by subtracting the value from the previous report from the current value. Each parameter also has a trend graph that provides a quick overview of the result. Fig. 2 shows a section from the report. A value displayed in red indicates an error that *must* be addressed. Warnings that *may* require investigation are shown in orange.

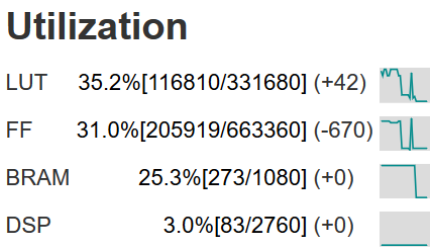

Figure 2: Excerpt of the utilization section from the `devkit_analyze` report.

The report contains the following sections:

Critical warnings

These warnings must be fixed, or at least, well understood.

CDC Critical

Check the `reports/post_impl.rpt` for details.

Synth Critical

Check the `reports/synth_runme.log` for details.

Impl Critical

Check the `reports/impl_runme.log` for details.

No Clock Critical

Check the `reports/post_impl_no_clock.rpt` for missing clock constraints.

Warnings

These warnings may cause problems and require investigation. Check the corresponding report file listed in the section for critical warnings for additional details.

Utilization

The total utilization of available instances of LUT, flip-flop (register), block RAM and DSP. This includes both the framework and user logic designs.

Timing

The importance of timing closure should be well known when working with an HDL designs. Any failing timing requirements must be fixed before uploading the firmware to the digitizer to avoid unexpected behavior.

Key values

These parameters are indicators that should be observed to avoid unwanted and/or unexpected results.

Congestion

Congestion is a measure of shortage of routing resources. A value of 5 or greater indicates shortage and will lead to significant problems with fulfilling the timing constraints. Please refer to *UltraFast Design Methodology* [2] for resolutions.

Maximum fanout

Fanout is the number of nodes driven by a register. The reported number is the maximum fanout in the design. Fanout are in most cases not a problem since Vivado automatically replicates registers with high fanout.

Note

Use of `KEEP`, `KEEP_HIERARCHY`, `DONT_TOUCH`, `MARK_DEBUG` or `ASYNC_REG` properties will prevent the replication of registers.

Control sets

A control signal is a signal to the set/reset and clock enable pin on a register. Registers that share a common control signal constitute a *control set*. The report provides the percentage of the recommended acceptable value given as a guideline in *UltraFast Design Methodology* [2]. Excessive usage may lead to difficulty in reaching the timing goals.

Route CPU time

Route CPU time is the process time used during route.

Place CPU time

Place CPU time is the process time used during place.

> **Note**
>
> An unexpected increase in the of place and/or route build time may indicate a potential problem in the design that may require intervention.

Excessive logic level

This report checks that the number of combinatorial instances between clocked primitives (registers in most cases) are reasonable with respect to the clock frequency. It is highly recommended to add pipelining to logic that is reported as “excessive” to avoid problems in timing closure. The following critical clocks are checked.

`pcie_axi_aclk`

AXI control bus clock. Check `reports/exlogic_pcie_axi_aclk.rpt` for details.

`clk_datapath`

Data path clock. Check `reports/exlogic_clk_datapath.rpt` for details.

`data_clk_2x`

Framework trigger logic clock. Check `reports/exlogic_data_clk_2x.rpt` for details.

`c0_dram_app_clk`

Framework DRAM logic clock. Check `reports/exlogic_c0_dram_app_clk.rpt` for details.

`c1_dram_app_clk`

Framework DRAM logic clock. Check `reports/exlogic_c1_dram_app_clk.rpt` for details.

> **Note**
>
> Under normal circumstances, the internal clocks should not report any *excessive logic levels*. However, due to Vivado’s optimization efforts, there may be reports if significant timing issues are introduced in the user logic. If the user logic is well timed and excessive logic levels are still reported, please contact TSPD support.

4 General Concepts

This section introduces concepts surrounding the development kit based on FWDAQ for ADQ3 series digitizers. Please refer to the *ADQ3 Series FWDAQ User Guide* [3] for the base knowledge on how the digitizer operates. The reader is assumed to be familiar with digital design in general and this section only serves to highlight some important aspects of digital design with respect to the development kit.

4.1 Parallel Digital Design

More often than not, the FPGA cannot be clocked at the same rate as the incoming data. To handle this scenario, the logic needs to be implemented to handle several data words per clock cycle, i.e. several data words in *parallel*. Parallel design is more challenging than its counterpart, where one data word is processed per clock cycle. Instructing the reader on parallel design is outside the scope of this document but moving forward, some familiarity with the concept is expected.

4.2 Data Flow

This section provides a brief description of each block in the data path. Fig. 3 shows an overview of the data path of FWDAQ for the ADQ3 series digitizers. The data propagates between the modules in the data path using a bus interface with custom insertion and extraction macros for convenience (see Section 5). Additionally, each module can be accessed from the AXI control bus. These two buses do *not* exist in the same clock domain, meaning any signals transferred from one domain to the other *must* be synchronized to the receiving clock. Refer to Sections 4.3 and 4.4 for additional details.

Important

Any signals transferred from the control bus clock domain to the data bus clock domain or vice versa *must* be synchronized to the receiving clock.

Trigger control

This module is tasked with inserting *events* on the data bus. The data bus has a single bit indicating the event itself and bits with additional information such as the position of the event within the parallel data words (samples).

Gain and offset

The digital gain and offset module is primarily intended for factory calibration but it may also be accessed by the user, and offers a way of scaling the signal.

Test pattern

This module may be configured to substitute the ADC samples with a synthetically generated pattern.

DBS

The digital baseline stabilizer, DBS, is designed for pulse signal measurement where high accuracy relative to a known baseline is required. Note that DBS is not enabled by default.

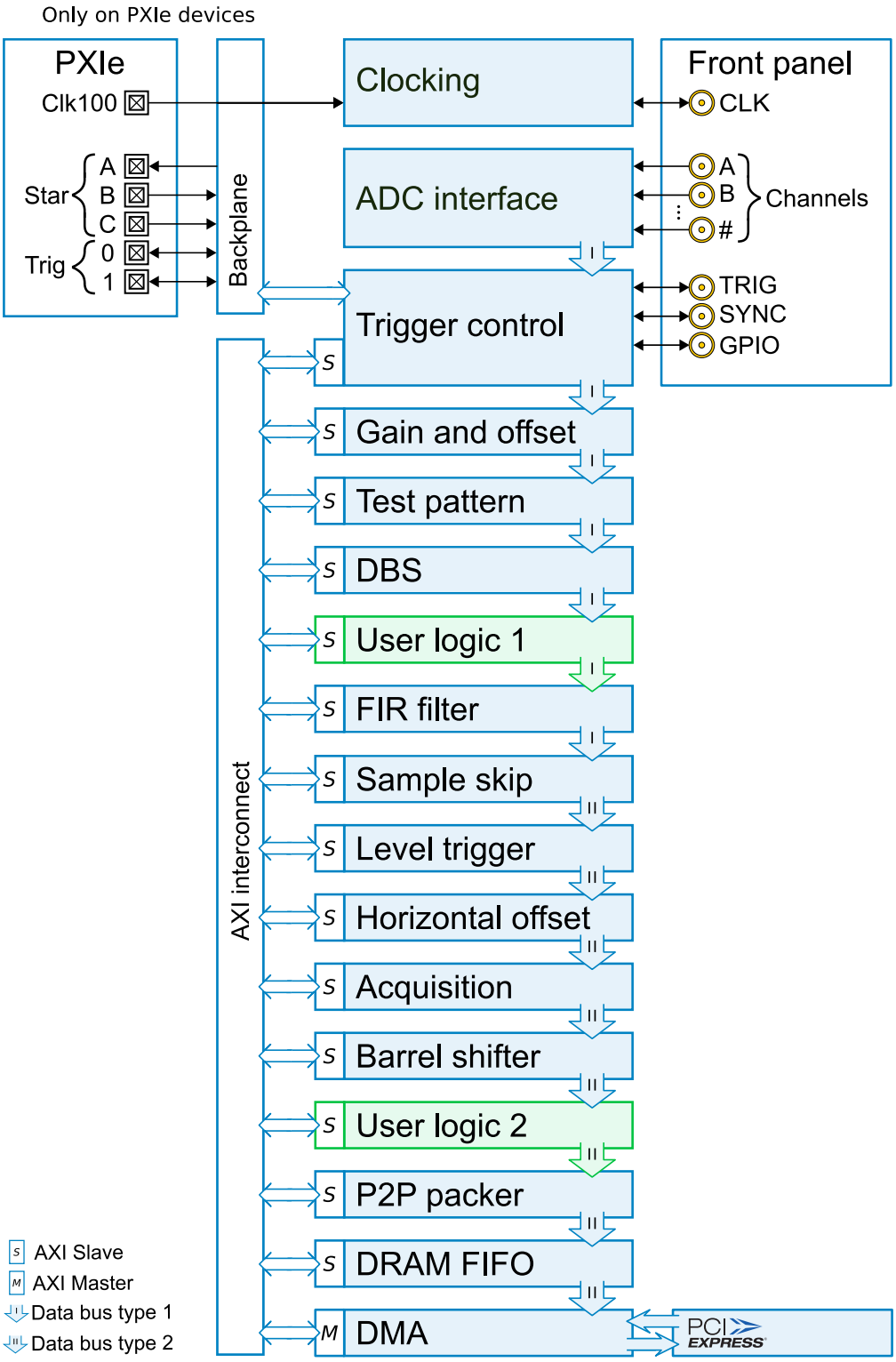

Figure 3: A block diagram of the data path of FWDAQ. The two user logic areas are highlighted in green.

User logic 1

This user logic area is intended to be used to process the incoming samples before any data reduction has been applied. Refer to Section 6 for a more detailed description of the module.

FIR filter

The base design contains a FIR filter module able to subject the data from each channel to a digital filter function. The FPGA resources used by this module can be reclaimed by the user. See Section 3.3.1 for more information.

Sample skip

The sampling rate can be reduced by the sample skip module. For example, setting the sample skip factor to 4 means that every fourth sample is kept and the others are discarded.

Level trigger

This module is used to insert events on the data bus based on the signal level within the data bus channels.

Acquisition

The acquisition module is tasked with framing the channels on the data bus into *records* based on a *trigger event*. The starting point and stopping point of the record is marked by a logic-high pulse on the record start and record stop data bus signals, respectively.

Barrel shifter

This module rearranges the parallel samples on the bus so that the first entry is the first sample in the record.

User logic 2

This user logic area is intended for processing data after the record framing and any data reduction has occurred. Refer to Section 7 for a more detailed description of the module.

Peer-to-peer packer

The peer-to-peer packer is tasked with converting the information on the data bus into packets that is suitable for the intended *endpoint* and application.

DRAM FIFO

Before the packets are dispatched to their endpoint, they pass through the on-board DRAM. This allows the digitizer to buffer packets in the event of a temporary stall on the physical interface or a short period where the acquisition rate is higher than the readout rate.

DMA

The DMA engine transports packets from the DRAM to a receiving endpoint.

4.3 Clock Domain Crossing Synchronization

A clock domain crossing, CDC, is a boundary where digital signals pass from one clock domain to another. This boundary constitutes a critical point in the design and care must be taken to *synchronize* signals

passing through the boundary to the *receiving* clock.

There are several techniques to choose from depending on the type of signal that should be synchronized, e.g. a multi-bit signal is not handled in the same way as a signal that is 1 bit wide. The reader is expected to be familiar with CDC synchronization techniques. The paper by Clifford E. Cummings [4] is a good place to start if the reader's knowledge needs to be refreshed.

In each of the user logic areas, there is one clock domain crossing in the default design—between the control bus clock and the data bus clock. This boundary joins the control bus register values, representing the current configuration, and the data bus logic, tasked with processing the data. To aid the user, there are two CDC helper modules available in the development kit framework, listed in Table 2.

Table 2: CDC modules available in the development kit framework.

Module	Description
`cdc_bit`	CDC synchronization module for a 1-bit signal (see Section 4.3.1).
`cdc_bus`	CDC synchronization of a multi-bit signal using a strobe (see Section 4.3.2).

These modules should cover most CDC needs and should be used whenever CDC synchronization is called for. Refer to Section 4.4 for details on the control bus and to Sections 6 and 7 for examples of CDC synchronization and logic making use of these register values.

4.3.1 CDC Synchronization of a 1-bit Signal

The module `cdc_bit` should *only* be used for a signal that is unrelated to other signals since the propagation of each bit may be different for parallel instances. Below is a description of the instantiation.

```
cdc_bit #(
   .ENABLE_OUTPUT_REGISTER("true") // String; "true" or "false".
                                   // Adds register on data output.
) cdc_bit_reset (
   .src_clk_i(src_clk),    // 1-bit input: Source clock.
   .dest_clk_i(dest_clk),  // 1-bit input: Destination clock.
   .src_data_i (src_data), // 1-bit input: Data driven by src_clk_i
   .dest_data_o(dest_data) // 1-bit output: Data syncronized to dest_clk_i
);
```

Important

In most cases it is recomended to keep the default value of `"true"` for the parameter `ENABLE_OUTPUT_REGISTER` since the registers used for the CDC synchronization will prevent the automatic replication performed by Vivado to reduce fanout. By adding the additional register, this situation is avoided.

4.3.2 CDC Synchronization of a Multi-Bit Signal

The module `cdc_bus` *must* be used when it is critical that each bit in a signal is propagated simultaneously.

```
cdc_bus  #(
   .WIDTH(WIDTH) // DECIMAL: Number of data bits.
) cdc_bus_datard (
   .src_clk_i(src_clk),   // 1-bit input: Source clock.
   .dest_clk_i(dest_clk), // 1-bit input: Destination clock.
   .src_rst_i(src_rst),   // 1-bit input: Reset dest_data to zero.
                          // NOTE: It is recommeded to tie src_rst_i low
                          // unless there is a specific reason to use it.

   .src_data_i(src_data),   // WIDTH-bit input: Data to be synchronized.
   .src_valid_i(src_valid), // 1-bit input: When asserted src_data_i will
                            // be synchronized to dest_clk_i.
                            // NOTE: This signal is ignored while
                            // src_ready_o is deasserted.
   .src_ack_o(src_ack),     // 1-bit output: Asserted when the data has
                            // been transferred to the destination clock.
   .src_ready_o(src_ready), // 1-bit output: While asserted the module is
                            // ready to accept new data.

   .dest_valid_o(dest_valid), // 1-bit output: Asserted when the data has
                              // been transferred to the destination clock.
   .dest_data_o(dest_data)    // WIDTH-bit output: Data synchronized
                              // to dest_clk_i.
);
```

Important

In most cases there is no need to clear the output registers in `cdc_bus` and it advised to tie `src_rst_i` low. It *shall not* be connected to a global reset as this will have negative impact on timing.

4.4 AXI Control Bus

Each user logic area can be accessed via the AXI control bus which provides a connection between the custom logic and the software running on the host computer. A transaction cannot be started from a user logic area and thus, communicating between the two modules using the control bus is not supported. Instead, transactions are initiated by the host interface which in turn is initiated from the ADQAPI, specifically by using the functions to read or write user registers:

- `ReadUserRegister()`
- `WriteUserRegister()`

Refer to the ADQ3 series user guide [3] for the API documentation.

Note

Transactions on the control bus cannot be initiated from the user design, only from calling specific functions in the ADQAPI.

4.4.1 Control Bus Signals

The control bus interface follows the AXI4-Lite format. The user logic needs to implement an AXI4-Lite slave instance.

4.5 Data Bus

The stream of ADC data, its associated control signals and other metadata all propagate on the data bus. The various signals are intricately related to each other and it is crucial that their relation in time is kept intact while they are processed by the custom logic.

Important

The bus signals are closely related to each other and it is crucial that their relation in time is kept intact through the user logic areas.

The development kit includes predefined functions to simplify the bus operations. There are two points where the user design interfaces with the data bus: *extraction* and *insertion*. As the names suggest, targeted signals are extracted from the bus and input to the custom logic to create a response. The logic's output signals are inserted back into the data bus and continues to propagate through the design (see Fig. 3). Signals that are *not* inserted back into the data bus will be subjected to pipelining with a delay equal to the value of the `BUS_PIPELINE` parameter. This parameter must be defined in the same HDL source file as the bus operations. Fig. 4 outlines the principle of working with the bus signals in the user space.

4.5.1 Two Bus Definitions

The design has two different data bus definitions that are labeled *data bus type 1* (DBT1) and *data bustype 2* (DBT2). The first user logic area uses the DBT1 bus definition while the second user logic area uses the DBT2 definition. The two bus definitions each have their own set of functions to support

bus operations. These are available in Verilog header (`.vh`) files in the `framework/` directory and are specified in Section 5. The user *must only* include the file matching the bus definition in the target user logic area, i.e.

- `databus_type1_functions.vh` for the first user logic area (UL1) and
- `databus_type2_functions.vh` for the second user logic area (UL2).

Note

In the default design, the source files for the two user logic areas includes the appropriate bus definitions.

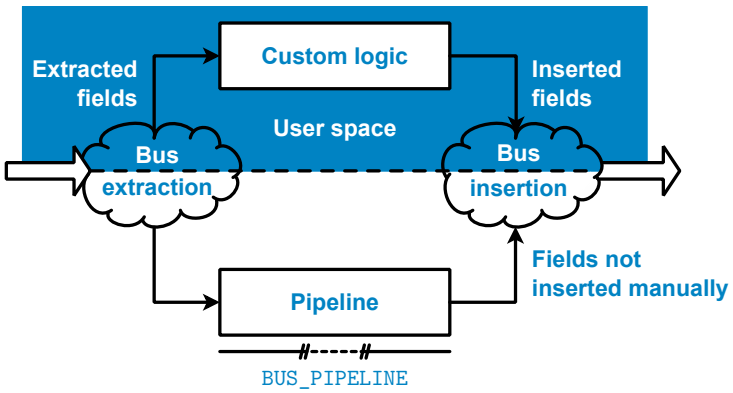

Figure 4: A diagram showing the principle of extracting signals from and inserting signals into the data bus. Any field not inserted manually is subjected to a pipeline delay equal to the value of the `BUS_PIPELINE` parameter.

5 Data Bus Signals

This section provides a reference for the bus signals that are present on the data buses. Each section defines a bus segment (which may consist of one or several signals) and provides a description of its functionality and purpose. The interface functions are named starting with one of the two prefixes `insert_*` or `extract_*`, to separate insertion and extraction. The functions are then further split into `common_*` and `channel_*`. *Common* signals are shared between all channels, with only a single entry on the bus, while the *channel* signals have one entry for each data channel. The functions to interface with the latter signal type take an additional argument to identify the target channel. Table 3 lists the documented bus signals and should be used as an index to navigate this section.

Example

To extract the common timestamp from the type 1 data bus (first user logic area), use

```
extract_common_timestamp(DONT_CARE)
```

where `DONT_CARE` can be any integer in practice since the argument is not used. For data bus type 2 (second user logic area), each channel has its own timestamp. For example,

```
extract_channel_timestamp(0)
```

extracts the timestamp from channel A and

```
extract_channel_timestamp(1)
```

from channel B and so on.

Example

To insert the common timestamp from the type 1 data bus (first user logic area), use

```
insert_common_timestamp(timestamp)
```

where `timestamp` is the 64-bit wide signal to insert. For data bus type 2 (second user logic area), each channel has its own timestamp. For example,

```
insert_channel_timestamp(timestamp, 0)
```

inserts the timestamp to channel A and

```
insert_channel_timestamp(timestamp, 1)
```

to channel B and so on.

Important

Bus signals not mentioned in this section are not yet implemented. Do not interact with these signals from the development kit.

Table 3: An overview of the data bus signals, including for which bus types the signals are present, and whether they are shared between the channels or separate.

Signal	Page	DBT1	DBT2
Timestamp	21	Common	Per channel
Timestamp synchronization	21	Common	Per channel
Trigger	22	Per channel	Per channel
Overrange indicator	23	Per channel	Per channel
General purpose	23	Per channel	Per channel
ADC sample data	23	Per channel	Per channel
Valid	23	N/A	Per channel
Record	23	N/A	Per channel
User ID	25	N/A	Per channel

5.1 Timestamp

The *timestamp* signal provides a monotonically increasing counter to serve as a time base for the digitizer. The signal is 64 bits wide and holds an unsigned value that may be synchronized to external and internal events by using the timestamp synchronization mechanism. The timestamp value during a data clock cycle where record start is asserted propagates to the user space in the host computer via the record header. The timestamp is *shared* between all channels for data bus type 1 and *per channel* for type 2.

5.2 Timestamp Synchronization

The *timestamp synchronization* bus segment is *shared* between all channels for data bus type 1 and *per channel* for type 2, just like the timestamp. The segment consists of three signals, described in the following sections.

5.2.1 Sample Index

The sample index is an unsigned value representing the integer part of the position of the timestamp synchronization event within the parallel sample data in a clock cycle. There is no fractional part, like in the trigger segment, so the value always indicates the index of the closest sample earlier in time. For example, if an event source with subsample precision is used to synchronize the timestamp, the position ‘3.6’ would propagate as ‘3’.

5.2.2 Count

The counter is tasked with keeping track of the number of timestamp synchronization events since the mechanism was last armed and is represented by an unsigned number. The counter value during a data clock cycle where record start is asserted propagates to the user’s application via the record header—provided the mechanism is set up and activated.

5.2.3 Event

The event is a 1-bit signal that, when asserted, indicates that a timestamp synchronization has occurred and that the other two fields are valid.

5.3 Trigger

The trigger bus segment is defined on a per-channel basis and consists of a few signals that together identify the trigger condition for the corresponding channel. The individual signals are described in the following sections.

5.3.1 Sample Index

The *sample index* signal is an unsigned value representing the integer part of the trigger position within the parallel sample data in a clock cycle. This field is concatenated with the trigger event sample index fraction field to get a fixed-point fractional value showing where the trigger occurred with full resolution. The value is only valid when the trigger event is asserted.

5.3.2 Sample Index Fraction

The *sample index fraction* signal is an unsigned value representing the fractional part of the trigger position within the parallel sample data in a clock cycle. The field is a fixed-point value in units of samples, where the decimal point is to the left of the most significant bit. This field is concatenated with the trigger sample index field to get a fixed-point fractional value showing where the trigger occurred with full resolution. This field is wider in DBT2 to maintain the trigger precision when using algorithms like *sample skip*, which increases the sampling period. The value is only valid when the trigger event is asserted.

5.3.3 Rising

The *rising* signal indicates the polarity of the trigger event. A rising edge event is indicated by a logic high level and a falling edge event is indicated by a logic low level. The value is only valid when the trigger event is asserted.

5.3.4 Event

The *trigger event* is a 1-bit signal indicating that the configured trigger condition has been met in this data clock cycle. The signal is active high.

5.3.5 Inhibit

The *inhibit* signal is controlled by the trigger blocking mechanism. The signal is one bit wide and a logic high level implies that triggers are *blocked*, i.e. trigger events are not converted into records by the acquisition module. Conversely, a logic low level implies that triggers are accepted.

5.4 Overrange

The overrange indicator is a 1-bit signal indicating that *at least* one sample within the parallel sample data in this clock cycle has saturated to the minimum or maximum value of the dynamic range.

5.5 General Purpose

The general purpose signal is 16 bits wide and may be used to achieve various firmware-specific goals. The default data acquisition firmware does not impose any restrictions on the signal, but special-purpose firmwares may reserve the signal for internal use. If no functional conflict exists, the signal may be used to, e.g. pass information between the two user logic areas in a well-defined manner. However, if the sample skip mechanism is active, only information passed while record start is asserted is preserved throughout the data path. Additionally, the value during a data clock cycle which asserts the record start bit, will propagate to the user space in the host computer via the record header.

5.6 Sample Data

The *sample data* signal holds the samples of the analog inputs on a per-channel basis and consists of several samples in parallel, as explained in Section 4.1. The width of the signal depends on the firmware configuration and target device. For this purpose, each user logic area defines constants which *must* be used to parametrize a custom design. For example, UL1 defines the width of one sample as `DBT1_CHANNEL0_BITS_PER_SAMPLE` and the number of parallel samples as `DBT1_CHANNEL0_PARALLEL_SAMPLES` for channel A. A sample is encoded using the *2's complement* representation. The data is MSB aligned, meaning that the MSB of the raw ADC data is located at bit index `DBT1_CHANNEL0_BITS_PER_SAMPLE-1` for channel A.

5.7 Valid

The *valid* signal exists on a per-channel basis on the type 2 data bus. The signal is one bit wide and when asserted, every sample within the parallel sample data in the current clock cycle is considered valid. For the type 1 data bus, the signal is not present since by definition, every clock cycle is considered valid.

5.8 Record

The record bus segment is only present on the type 2 data bus and consists of the signals that frame a record. The following section describe the individual signals.

5.8.1 Start, Start Index, Stop and Stop Index

The two 1-bit signals *record start* and *record stop* work together frame a record of sample data. They are *inclusive*, meaning that if either signal is asserted, the sample data associated with that data clock cycle belongs to the record. The signals *may not* be asserted in the same data clock cycle.

Important

The two 1-bit signals record start and record stop may not be asserted in the same data clock cycle. Thus, the minimum record length is two data clock cycles.

The two index signals define the start and stop positions within the parallel data word. Both indexes are inclusive. A record starts at the sample indicated by *record start index* and stops at the sample indicated by *record stop index*.

The record start and stop signal must be sent on a clock cycle where the valid signal is asserted to have any effect. Any custom user logic should always check that the valid signal is asserted before reacting to the record start signal. Note that record start and record stop signals may be asserted for multiple cycles as long as one—and only one—clock cycle is overlapping with the valid signal. Fig. 5 presents a timing diagram where the valid signal is always asserted. In Fig. 6, data is reduced by the user logic module. Note that the is no requirement for the valid signal to be periodic, it may be asserted and deasserted as needed.

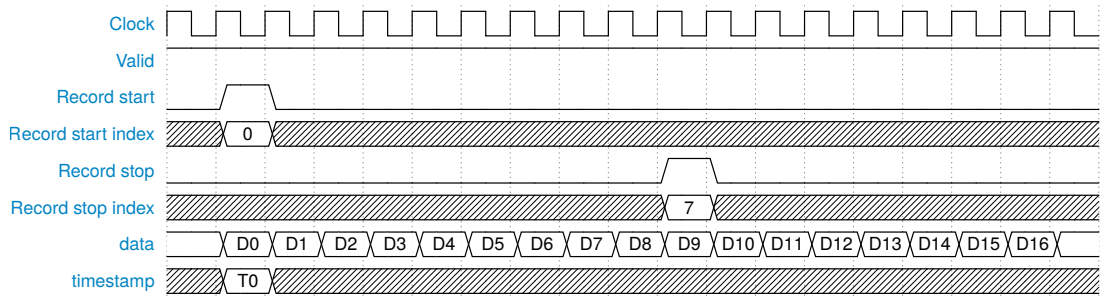

Figure 5: Timing diagram for the record framing signals when no data reduction is active. The record spans ten data clock cycles.

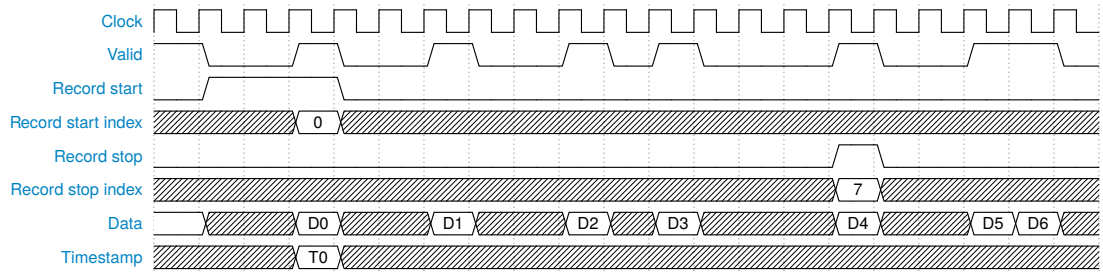

Figure 6: Timing diagram for the record framing signals with data reduction active. The record spans five valid data clock cycles durring thirteen clock cycles.

It is important that the integrity of the record bits is preserved. The second user logic area must output one—and only one—record stop event for each record start event. To discard a record, record start and record stop assertions must be removed *in pairs*. If multiple record start events are output without their corresponding record stop events, data corruption will ensue. Note that it is valid to create an infinite stream of data by generating a single record start event. Listed below is a summary of the properties of the record bits.

- Only one record stop event per record start event may be output. Multiple record start events without any record stop events will result in data corruption.
- The record bits are only valid when valid is asserted.
- The record bits may not be asserted in the same data clock cycle.
- The record bits may be asserted for multiple cycles. Only one of these cycles must have data valid asserted.

5.8.2 Number

The record number is an unsigned value that keeps track of the number of records that have been created since the data acquisition was last started. The value is only valid when record start is asserted and wraps at the maximum value.

5.9 User ID

The user ID is an 8-bit signal with similar function to the general purpose signal. However, this signal may never be claimed for firmware-specific purposes and is reserved for the user. The value during a data clock cycle in which record start is asserted will propagate to the user space in the host computer via the record header.

5.10 Differences relative to ADQ14, ADQ7 and ADQ8

This section provides a short list of notable differences in the ADQ3 series digitizers data buses compared to previous digitizers is provided to help with migrating a design.

- The data buses are now named *data bus type 1* (DBT1) and *data bus type 2* (DBT2) instead of *real-time* and *reduced rate*.
- *Low-rate channels* have been removed, each channel now instead has parameterized number of parallel samples and data width.
- Insert/extract function naming scheme has been changed completely to be more consistent.
- The record start index and record stop index signals have been added to allow for a record length granularity of single samples. This also means that a fixed record length can now result in a varying number of clock cycles between start and stop, depending on which sample the record starts at.

6 User Logic 1

The first user logic area (defined in `user_logic1.v`) uses the type 1 bus definition (Section 4.5) for its incoming and outgoing data bus. At this point in the data path (Fig. 3), there is *no* data valid signal present since every data clock cycle is considered valid. Thus, any custom logic placed in this area must be designed to output valid data on each data clock cycle.

Important

The first user logic expects valid data to be output on each data clock cycle. There is no data valid signal at this point in the data path.

Note

The file `framework/dbt1_param.vh` defines constants to use when parametrizing a design in the first user logic area.

6.1 Default Contents

By default, the first user logic area contains an example of how to interact with the data bus, how to perform CDC synchronization and a simple test pattern generator. When activated, the test pattern generator replaces the channel data with a monotonically increasing ramp starting from the base value specified in BASEVALUE and incrementing by one for each sample in the parallel data word. The pattern restarts in the next data clock cycle.

Example

With a base value of 1000 and a parallelization of 8, the generator will output

```
1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1000, 1001, 1002, ...
```

6.2 Register File

The register file for the first user logic area is defined in the file `user_logic1_s_axi.v`. The default implementation provides an interface consisting of a read port and a write port to each 32-bit register separately. The naming follows the perspective of the host computer, i.e. the read port provides the host computer with data from the firmware and vice versa for the write port.

Adding a new register to the default implementation involves declaring a new set of matching ports and modifying the logic (in `user_logic1_s_axi.v`) to target this new register when a matching bus transaction occurs. Refer to the existing design for implementation details.

Important

Failing to implement correct handling of the AXI bus transactions in `user_logic1_s_axi.v` can cause the host computer to hang when the offending register is accessed.

The register access is controlled in the module instantiating the register file, i.e. (in `user_logic1.v`). To construct a register with read and write access, connect the two corresponding ports of the register

file, creating a loopback connection. To instead construct a read-only register, ignore the write port and provide data for the read port. Finally, to construct a write-only register, tie the read port to a constant value. Note that registers with write-only access cannot be accessed using a masked write, since the masking operation relies on being able to read the register's contents. Refer to the existing design for implementation details and examples.

6.3 Default Register File

This section documents the register file for an unmodified version of the development kit.

Name UL1ID

Offset `0`

Default `0x00abcdef`

This register contains a constant value to identify the first user logic area.

31	30	29	28	27	26	25	24
ID[31:24]							
ID[23:16]							
ID[15:8]							
ID[7:0]							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31:0 – ID: Constant identification value *Read-only*

This range always reads as `0x00abcdef` with the default user logic contents. However, the value can be changed by modifying the HDL design.

Name UL1CONTROL

Offset `1`

Default `0`

This register contains the control bit for the simple test pattern generator described in Section 6.1.

31	30	29	28	27	26	25	24
TESTEN	-						
-							
-							
-							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31 – TESTEN: Test pattern enable *R/W*

If set to 1, the default user logic will replace the data for each channel with a simple test pattern, see Section 6.1 for additional details. The default value is 0, leaving the data for each channel unmodified.

Name UL1BASEVALUE

Offset 2

Default 0

This register contains the base value used by the simple test pattern generator described in Section 6.1.

31	30	29	28	27	26	25	24
BASEVALUE[31:24]							
BASEVALUE[23:16]							
BASEVALUE[15:8]							
BASEVALUE[7:0]							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31:0 – BASEVALUE: Test pattern base value *R/W*

The base value for the simple test pattern controlled via TESTEN. While the range is 32 bits wide, only as many bits as the channel specifies will be sliced from the register, starting at the least significant bit. The resulting range will be interpreted as a 2's complement value.

7 User Logic 2

The second user logic area (defined in `user_logic2.v`) uses the type 2 bus definition (Section 4.5) for its incoming and outgoing data bus. At this point in the data path (Fig. 3), there is a data valid signal present and the user may modify the output data stream by modulating this signal. However, creating records of varying sizes, is currently not supported.

Important

Creating records of varying sizes is currently not supported. Each record output by a specific channel must be the same length. However, the record length does not need to be the same value for all channels.

Important

It is crucial that the record framing signals output from the second user logic area have the correct behavior with respect to the data valid signal.

Note

The file `framework/dbt2_param.vh` defines constants to use when parametrizing a design in the second user logic area.

7.1 Default Contents

By default, the first user logic area contains an example of how to interact with the data bus, how to perform CDC synchronization and a simple test pattern generator. When activated, the test pattern generator replaces the channel data with a monotonically increasing ramp starting from the base value specified in BASEVALUE and incrementing by one for each sample in the parallel data word. The pattern restarts in the next data clock cycle.

Example

With a base value of 1000 and a parallelization of 8, the generator will output

```
1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1000, 1001, 1002, ...
```

7.2 Register File

The register file for the second user logic area is defined in the file `user_logic2_s_axi.v`. The default implementation provides an interface consisting of a read port and a write port to each 32-bit register separately. The naming follows the perspective of the host computer, i.e. the read port provides the host computer with data from the firmware and vice versa for the write port.

Adding a new register to the default implementation involves declaring a new set of matching ports and modifying the logic (in `user_logic2_s_axi.v`) to target this new register when a matching bus transaction occurs. Refer to the existing design for implementation details.

Important

Failing to implement correct handling of the AXI bus transactions in `user_logic2_s_axi.v` can cause the host computer to hang when the offending register is accessed.

The register access is controlled in the module instantiating the register file, i.e. (in `user_logic2.v`). To construct a register with read and write access, connect the two corresponding ports of the register file, creating a loopback connection. To instead construct a read-only register, ignore the write port and provide data for the read port. Finally, to construct a write-only register, tie the read port to a constant value. Note that registers with write-only access cannot be accessed using a masked write, since the masking operation relies on being able to read the register's contents. Refer to the existing design for implementation details and examples.

7.3 Default Register File

This section documents the register file for an unmodified version of the development kit.

Name UL2ID

Offset `0`

Default `0x12345678`

This register contains a constant value to identify the first user logic area.

31	30	29	28	27	26	25	24
ID[31:24]							
ID[23:16]							
ID[15:8]							
ID[7:0]							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31:0 – ID: Constant identification value *Read-only*

This range always reads as `0x12345678` with the default user logic contents. However, the value can be changed by modifying the HDL design.

Name UL2CONTROL

Offset `1`

Default `0`

This register contains the control bit for the simple test pattern generator described in Section 7.1.

31	30	29	28	27	26	25	24
TESTEN	-						
-							
-							
-							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31 – TESTEN: Test pattern enable *R/W*

If set to 1, the default user logic will replace the data for each channel with a simple test pattern, see Section 7.1 for additional details. The default value is 0, leaving the data for each channel unmodified.

Name UL2BASEVALUE

Offset 2

Default 0

This register contains the base value used by the simple test pattern generator described in Section 7.1.

31	30	29	28	27	26	25	24
BASEVALUE[31:24]							
BASEVALUE[23:16]							
BASEVALUE[15:8]							
BASEVALUE[7:0]							
7	6	5	4	3	2	1	0

Range Descriptions

Bit 31:0 – BASEVALUE: Test pattern base value *R/W*

The base value for the simple test pattern controlled via TESTEN. While the range is 32 bits wide, only as many bits as the channel specifies will be sliced from the register, starting at the least significant bit. The resulting range will be interpreted as a 2's complement value.

8 Timing Closure

In the event of timing closure problems, please follow these guidelines:

1. Run `devkit_analyze` as described in Section 3.6 and fix any reported issue. Pay special attention to resolve any *excessive logic level*.
2. Review the failing paths and surrounding logic in Vivado's schematic viewer.
3. Carefully review that you conform with the recommendations in *Timing ClosureQuick Reference Guide (UG1292)"* [5]. For detailed information see *UltraFast Design Methodology User Guide Guide (UG949)* [2].

9 Troubleshooting

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

If the error message is difficult to interpret, the Teledyne SP Devices support can be reached via e-mail at spd_support@teledyne.com. Make sure to include a trace log file from a run where the error appears.

However, the support team *cannot* help the user with issues originating in the user's custom design in any of the user logic areas. Additionally, no training sessions on the topic of HDL design will be offered free of charge.

When facing a problem localized to the custom user logic design, Section 9.1 provides one possible way forward in those situations.

Important

Teledyne SP Devices' support cannot help with issues localized to the user's custom logic design nor offer training for HDL design concepts.

9.1 Debugging on Hardware

The section describes *one* possible workflow for setting up and connecting to a Xilinx debug core. Refer to the Xilinx documentation for further instructions. A good starting point is the *Vivado Programming and Debugging User Guide* [6].

Warning

Debugging on hardware requires physical access to the JTAG port on the digitizer PCB.

9.1.1 Creating the Debug Core

1. Mark the signals as debug with the `mark_debug` property, for example in Verilog:

   ```
   (* mark_debug = "true" *) wire signal_to_debug;
   ```

   Setting the `mark_debug` property makes the signals available in the debug wizard and ensures that the tool will not remove the signals in optimization.

2. Synthesize the design by clicking on *Run Synthesis* and wait for Vivado to finish synthesizing the complete design.

3. Open the synthesized design by clicking on *Open Synthesized Design*.

4. Open the debug wizard by clicking on *Setup Debug* and follow the instructions.

5. Close and save the synthesized design. When asked for a target file to write constraints to, choose to create a new file, to avoid affecting the constraint files of the development kit framework.

6. Under the *Design Runs* tab, if the `synth_1` step is listed as out of date, right click and choose *Force Up-To-Date*.
7. Generate the bitstream by clicking on *Generate bitstream*.
8. When the process has finished, run the Tcl command

```
devkit_mcs
```

9. The generated files are found in
   - `artifacts/latest/devkit.ltx`
   - `artifacts/latest/devkit.mcs`
   - `artifacts/latest/devkit.bit`
10. Program the firmware image (`.mcs` file) using ADQUpdater. Refer to the ADQUpdater user guide for instructions on how to manage the firmware on the ADQ3 series digitizer [1].

9.1.2 Connecting to the Debug Core

The *Vivado hardware manager* is used to connect to the debug core. Connecting to the debug core requires:

- that the `.mcs` file with core has been programmed; and
- that the `.ltx` file is available.

Depending on the clock signals chosen for the debug core, the firmware may have to be initialized before the Vivado hardware manager can find the debug core. For details on initialization, refer to the user guide for the ADQ3 series user guide [3].

Important

The clock used for the debug core must be running for the core to function.

1. Connect the Xilinx platform cable to the digitizer's JTAG port.
2. Start Vivado and click on *Open Hardware Manager*.
3. Click on *Open Target* and chose *Auto Connect*.
4. In the trigger setup window, click on *Specify probe file and refresh device*.
5. Browse to the `debug_nets.ltx` file and click on refresh.

Refer to the *Vivado Programming and Debugging User Guide* [6] for further instructions.

References

[1] Teledyne Signal Processing Devices Sweden AB, *18-2059 ADQUpdater User Guide*. Technical Manual.

[2] Xilinx Inc., *UltraFast Design Methodology*, August 2020. User Guide Guide (UG949).

[3] Teledyne Signal Processing Devices Sweden AB, *21-2539 ADQ3 Series FWDAQ User Guide*. Technical Manual.

[4] C. E. Cummings, “Clock domain crossing (CDC) design & verification techniques using SystemVerilog,” in *SNUG 2008 proceedings*, (Boston, MA, USA), Sunburst Design, Inc., 2008.

[5] Xilinx Inc., *UltraFast Design Methodology Timing Closure*, June 2020. Quick Reference Guide (UG1292).

[6] Xilinx Inc., *Programming and Debugging*, June 2020. User Guide (UG908).

TELEDYNE SP DEVICES
Everywhereyoulook™

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.