TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ7 Development Kit

User Guide

Author(s): Teledyne SP Devices
Document ID: 17-2010
Classification: Public
Revision: PB3
Print date: 2019-03-12

Contents

1 Introduction

This document describes the usage of the ADQ7 Development Kit. Note that there are different versions of the Development Kit depending on what version of ADQ7 that is targeted. There are also different versions for one- and two-channel operation mode. Please make sure that you are using the version that matches your hardware and needs.

1.1 Definitions and Abbreviations

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

Table 1: Definitions and abbreviations used in this document.

Term	Explanation
ADC	Analog-to-digital converter
AFE	Analog front-end
HDL	Hardware description language
µB	Microblaze (Xilinx soft processor)

2 Prerequisites

- A license for the ADQ7 Development Kit, purchased from Teledyne SP Devices.
- A license of the Xilinx Design Tools. For current version of the ADQ7 Development Kit a license of Vivado 2017.1 is required. (see table below)
- The Vivado license includes simulation tools for mixed VHDL/Verilog code.
  - Minimum required is Design Edition.
  - WebPack does not support the ADQ boards.
  - Xilinx ISE cannot be used for ADQ7.
- To create custom logic you need skills in Verilog or VHDL design.

Table 2: Version requirement

Development Kit Revision	Tool version
>r33165	Vivado 2017.1

3 How to use the Development Kit

This section describes the setup and typical design flow

3.1 Extracting the DevKit files

Extracting the devkit .zip archive yields the folders seen below

Archive root

- **constraints**
- **documentation**
- **edif**
- **elf**
- **implementation**
- **ip**
- **source**

Figure 1: Devkit .zip file content

The source files available for editing are

- user_logic1.v
- user_logic2.v
- ul1_regfile.v
- ul2_regfile.v

These files are found in the source folder. Do not modify any other files.

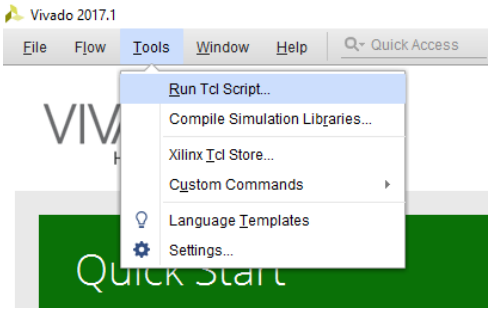
- Pre-compiled code is placed in the edif directory.
- The ip directory contains Vivado IP cores
- The scripts used setup the project in Vivado and run the build process are placed in the implementation/scripts folder

Note

Extraction of the archive must be performed in a directory where you as a user have permission to write and change files. (For instance *not* under `Program Files/SP Devices`)

3.2 Open the Development Kit

1. Start Vivado (it can be found under Xilinx Design Tools in the start menu after installation)
2. In the Vivado menu select *Tools / Run Tcl Script*



3. Select the file: `devkit/implementation/scripts/devkit.tcl`. The Tcl Console will show the message below.

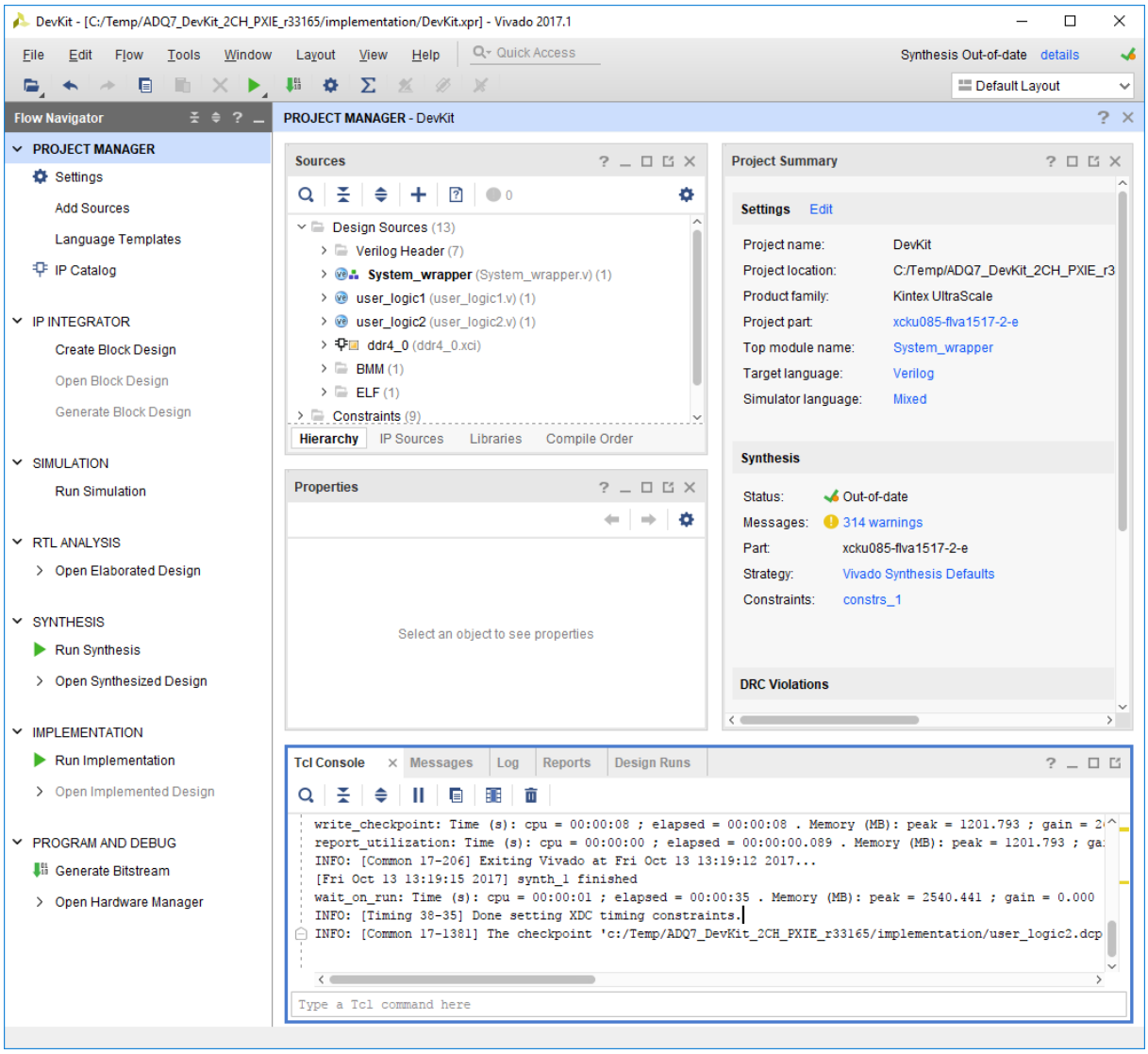
```
*** ADQ7 Development Kit ***
Usage:
  devkit_setup          - Create project
  devkit_build          - Build project
  devkit_synth_ul 1     - Generate netlist for User_Logic1
  devkit_synth_ul 2     - Generate netlist for User_Logic2
  devkit_mcs            - Generate .mcs firmware file
```

3.3 Set up the project

Go to the Tcl Console command field and type:

```
devkit_setup
```

and press Return. This will create the Vivado project. The initial setup will take a moment since parts of the design will be compiled. When the execution has finished a project has been created.



Note

The user logic modules must be compiled into netlists. This is done by the provided scripts. Using the normal flow in Vivado instead of the provided scripts will not work.

3.4 Build the firmware bitfile

Go to the Tcl Console command field and type:

```
devkit_build
```

and press Return. Dependent on your computer specifications and the complexity of the total logic (pre-compiled + your user logic) this may take between 1 hour to 10 hours. When the execution has finished, an *.mcs file that can be flashed to the digitizer have been created in the implementation folder.

You can manually rebuild the netlist for the user logic modules with the Tcl commands:

```
devkit_synth_ul 1
devkit_synth_ul 2
```

Then use the Vivado GUI to generate the bitstream. To convert the bitstream to an .mcs file use the Tcl command:

```
devkit_mcs
```

Important

Due to a bug in Vivado you must open the design after synthesis and run the tcl command refresh_design. After that you can continue with the implemtation step in the Vivado GUI. This is automatically done when running devkit_build.

3.5 Working with your design

You can use the command below to set your user logic as top module. This is useful when using the Vivado RTL analysis tools. Then use

```
devkit_set_top_ul 1
devkit_set_top_ul 2
```

You can go back to the standard devkit top module with the command:

```
devkit_set_top
```

3.6 Typical DevKit design flow

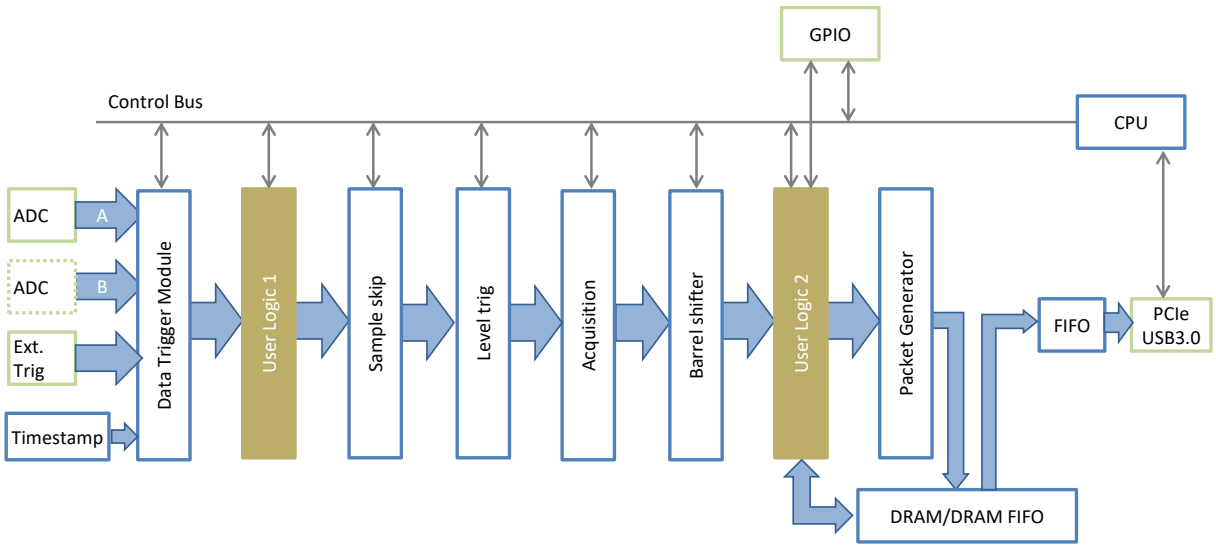
1. Set up the DevKit Project as described in Section 3.3.
2. Modify or insert new verilog code into user_logic1.v or user_logic2.v. This can be subdivided into 4 steps:
   (a) Extract data, trigger and data_valid signals using the extract macros.
   (b) Process the extracted data and signals according to your requirements.
   (c) Insert the processed data, trigger and data_valid signals back into the data path.
   (d) Set the correct BUS_PIPELINE delay to keep bus signals which were not manually inserted, in sync.
3. Either run devkit_build to generated the configuration file (.mcs) or if you prefer to do it manually follow the steps below.
4. Generate the netlist for the modified code by running:
5. devkit_synth_ul 1 and/or devkit_synth_ul 2
6. Run Synthesis by clicking Run Synthesis in Vivado

7. Open the design by click Open Synthesized Design
8. When the design has opened run the Tcl command: `refresh_design`
9. Click on Generate Bitstream
10. Wait for implementation to finish.
11. Generate a .mcs file to load into the onboard flash of the ADQ7 by running: devkit_mcs in Vivado Tcl console. The generated .mcs file can be found in the `implementation` folder.
12. Load the newly generated custom firmware file (.mcs) into the ADQ7 by using the ADQUpdater application, available via the SDK
13. Test the custom firmware using the ADQAPI library available via the SDK, using one of the many software examples as a basis for your software application

4 Basic design with the Development Kit

4.1 Dataflow chart

The figure below illustrates the data flow path of ADQ7. The user logic modules are marked.



4.2 Control bus

Each User Logic modules has its own register control bus. The example code implements a register bank. The bus can also be used to interface block RAMs, FIFOs, or other custom blocks. The control bus signals are listed in Table 3.

Important

The first four 32-bit words are reserved for internal functions and cannot be used.

Table 3: Control bus signals

Signal	Description
clk	CPU Clock
rst_i	Active high startup reset
addr_i	Read/Write address 14 bits
wr_i	Active high write strobe
wr_ack_o	Active high write data qualifier
wr_data_i	Write data 32 bits
rd_i	Active high read strobe
rd_ack_o	Active high read data qualifier
rd_data_o	Read data 32 bits

4.2.1 Accessing the block control bus from the ADQAPI

The following API commands can be used to access the user logic registers

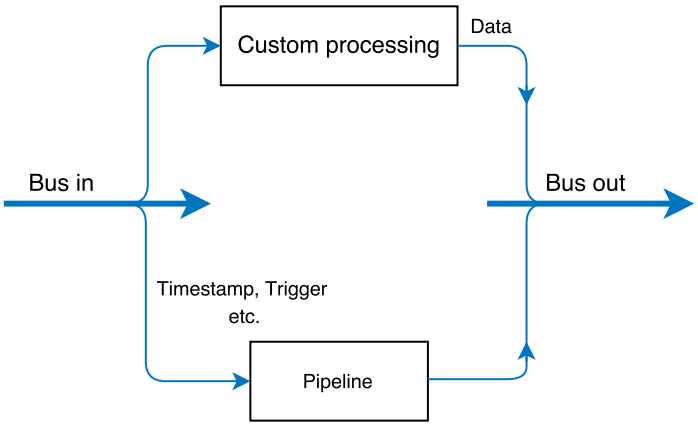
- Access single register:
  - `ReadUserRegister()`
  - `WriteUserRegister()`
- Write range of addresses:
  - `ReadBlockUserRegister()`
  - `WriteBlockUserRegister()`

For details see the ADQAPI_ReferenceGuide.pdf (Provided by the SDK installer)

4.3 Data stream bus

To keep information consistent through the digitizer system, the ADC data, trigger data and timestamp are provided as a combined stream bus. Therefore it is important that all these signals have the same latency through the User Logic designs.

There is a bus splitter macro in the user logic code that helps you to do this in a simple manner (see next section).



4.4 Concept of parallel samples (parallel design)

The FPGA can not be clocked at the sampling rate of the digitizer channels. A clock frequency on the FPGA of 200–400 MHz is a good choice, to avoid timing issues. The ADQ7 unit has a sampling rate of 10 GSPS or 5 GSPS this means we need to have 32 or 16 parallel samples for each clock. The user logic will receive the parallel samples and will need to output the parallel sample on each clock at 312.5 MHz. This clock signal is denoted as an input port named `s_axis_aclk` or `m_axis_aclk` in the user_logic module and should be used to clock any custom data manipulation block.

Note

The clock frequency is 312.5 MHz for the data stream. Either s_axis_aclk or m_axis_aclk can be used.

Table 4: Data bus parallelization for ADQ7

Digitizer	Channels	Sample rate	Parallelization
ADQ7	1 Channels	10 GSPS	32 samples per cycle
ADQ7	2 Channels	5 GSPS	16 samples per cycle

4.5 Example of the bus splitter macro usage.

ADC data from channel A is extracted from the bus and delayed one clock cycle. The signals from the bus that are not used are automatically passed through the module, with a delay equal to the `BUS_PIPELINE` parameter, which is set to 1 to match the data delay.

Listing 1: Data bus extraction and insertion

```
localparam BUS_PIPELINE = 1;

always @(posedge clk_data) begin
  // ...
```

```
  data_a <= extract_ch_all(CH_A);
  // ...
end

// User inserting into bus output
always @(*) begin
  init_bus_output();
  // ...
  insert_ch_all(data_a, CH_A);
  // ...
  finish_bus_output();
end
```

Important

Setting BUS_PIPELINE value correctly is critical to maintain valid data in the digitizer framework

4.6 Trigger in User Logic 1

Each data channel in the bus has an associated trigger vector that can contain external trigger, software trigger, internal trigger etc. The trigger mode selection (software, internal, external, etc) only decides which data is inserted into the trigger vector at the start of the data path, before the user logic modules. If you want to create your own trigger, you can simply ignore the incoming trigger vector to the user logic, and create a new trigger vector of your own which you insert onto the outbound data bus.

4.7 Commonly used bus splitter functions

Table 5: Commonly used bus splitter functions

Output bus	Input bus	Description
`init_bus_output()`	N/A	Must be placed **before** the `insert_` functions
`finish_bus_output()`	N/A	Must be placed **after** the `insert_` functions
`insert_ch_all(data, ch)`	`extract_ch_all(ch)`	Data for channel `ch`
`insert_data_valid(dv, ch)`	`extract_data_valid(ch)`	Data valid bit
`insert_over_range(ovr, ch)`	`extract_over_range(ch)`	Over range bit

The bus splitter functions are defined in the header files

- `bus_splitter_rr.vh` for the reduced rate bus (User logic 2)
- `bus_splitter_rt.vh` for the real time bus (User logic 1)

which have to be included in the module together with the `device_param.vh` file. The fields on the data bus is listed in Table 6/

Table 6: Data bus fields

Field	Comment
Data valid	Per channel
General purpose vector	Per channel
User ID	Per channel
Over range	Per channel
Gate count	Per channel
Record bits	Per channel
Trigger vector	Per channel
Data	Per channel
Aux trigger	
Timestamp	

4.8 Handling of the trigger vector

Each data channel has an associated trigger vector in the bus. The trigger information can be extracted by using the macro shown in the example below.

The trigger vector in User Logic 2 has 16 additional fractional bits compared to User Logic 1. This is to allow maintaining the full trigger precision when using sample skip or decimation.

When using the standard firmware, the trigger vectors for all channels are identical. Other firmware packages such as FWPD will let the channels operate with individual triggering (such as individual level trigger).

The trigger data is inserted prior to the user logic module and is selected from a number of trigger sources via the SetTriggerMode API command. If you want to create your own trigger, you can simply ignore the incoming trigger vector to the user logic, and create a new trigger vector of your own which you insert onto the outbound data bus. Trigger vector bus width:

- User Logic 1: 7 bits
- User Logic 2: 23 bits

The trigger vector is divided into 3 fields. That can be extracted with macro functions. trigger vector:

4.9 Trigger bus splitter functions

The trigger position value should be interpreted as a fixed-point fractional number in units of samples in the current sample rate. The value is in the format {5.3}. For example if the value is 8'b00100100 this means that the trigger is at position 4.5 samples in the current clock cycle of data. The event bit indicates

Table 7: Trigger bus splitter function

Output bus	Input bus	Description
`insert_ch_trig_trising(rising, ch)`	`extract_ch_trig_trising(ch)`	Trigger polarity bit
`insert_ch_trig_tevent(event, ch)`	`extract_ch_trig_tevent(ch)`	Trigger event bit
`insert_ch_trig_tnum(num, ch)`	`extract_ch_trig_tnum(ch)`	Trigger position

`ch` is the channel index, starting from 0

whether there was a trigger event (1) or not (0) during the current clock cycle. The polarity bit is used to indicate rising(1)/falling(0) edge. It does not affect anything in the data acquisition.

If you are using multirecord data collection, it is required that the trigger vectors for all channels are identical. If you are using triggered streaming, they can be made different.

5 Advanced design with the development kit

This section describes the advanced features of the ADQ7 development kit.

5.1 The inner design of the Multiport DRAM

Multiport is essentially a multiple port interface towards the two DRAM controllers. It handles port arbitration, DRAM command generation and allows both read and write ports. The user logic 2 in ADQ7 has access to one writer port and one reader port on each of the two DRAM conntrolles, while the framework design at the same time also has a number of reader and writer ports.

The writer ports and reader ports respectively, share the same structure and have the same functionality. The only difference is how they are prioritized in their access to the DRAM.

Important

The DRAM is by default used as a FIFO for the data transfer to the PC. This FIFO must be disabled before acessing the DRAM from user logic 2. This is done by the ADQAPI command:
SetStreamConfig()
Please be aware of that the data transfer will in this case only use a small FIFO and that this may cause data owerflow if data is generated faster than the readout to the PC.

5.1.1 Ports

Since multiport handles the port arbitration, it is also the master on both the reader port and writer port buses, i.e. it signals *I will read data this clock cycle* on the writer port, and *I am outputting valid data this clock cycle* on the reader port.

The memory space which is to be read from / written to is selected by the device communicating with the port, via address pointers and a strobe signal.

There are no FIFOs in the actual ports, they are effectively just interfacing between the FIFO in the device using the port, and the command/data FIFOs.

Something that should be noted is that the ports themselves run on the global memory clock in the FPGA, but the DRAM controller runs on its own DRAM clock. These run at the same clock rate but are separate clock networks. For write operations, the clock domain crossing happens in the command/data FIFOs. For read operations, there is no such FIFO in multiport, however, and the data is instead just clocked directly to the memory clock domain.

Both reader and writer ports support address wrapping. In the case of the writer port, the write address will keep wrapping from last to first address, until the write_last signal is asserted.

In the reader port there are two sets of addresses: high and low set up a memory area to wrap around, while first and last set up the start address and end address of the readout. Since the digitizers often use circular writing to memory areas until a trigger occurs, the typical use case is for the reader port is to set high/low to the edges of the circular buffer, set first to wherever the trigger.

5.1.2 Command mux and port arbitration

The command mux selects which port is allowed to input commands to the command FIFO. The mux contains state machines called `select` and `select_hot`, which are actually duplicates of each other but with different encoding (integer coded and one-hot coded respectively) in order to improve timing. These are used to select which port is current enabled.

There is a strict prioritization between ports (see the ordering in the overview block diagram). As soon as a higher priority port signals *not empty*, the `select` register changes value and the mux starts accepting command from the new port starting with the next clock cycle.

5.1.3 Command / data FIFO

Data is read from / written to the DRAM using commands. The current multiport module only supports memory controller setups which produce one clock cycle of data per command.

As an example, the ADQ7 memory architecture has a 64-bit external bus, with a 1:4 memory controller giving 256 bits internally. The burst setting is also 1:4, resulting in a burst of four 64-bit accesses for each command, giving a single cycle of internal 256-bit data.

The ports automatically generate read/write commands across the space which the communicating device requested via strobe and address inputs

5.1.4 Tag FIFO

A write that is sent to the writer FIFO has no need to keep track of which port sent the write. A read however, needs to know where to send its results. That is what the tag FIFO is used for. At the same time as a command is sent to the command FIFO, a read port address is also entered into the tag FIFO. After the command has been sent, and the data returned from the DRAM, the tag is used to determine which port to send the read data to.

The tag FIFO also passes two additional bits, firstdata and lastdata, which are generated by the reader port to signal which data words are first and last in a read operation.

5.2 DRAM User interface

Table 8: DRAM User interface

Signal name	Direction	Description
read_reset_i	Input	Reset signal
read_strobe_i	Input	Strobe in order to start a read operation
read_abort_i	Input	Assert to abort read operation (stop generating read commands)
read_first_addr_i	Input	First address of read operation
read_last_addr_i	Input	Last address of read operation
read_low_addr_i	Input	Low address of read operation (memory wrap)
read_high_addr_i	Input	High address of read operation (memory wrap)
read_sent_o	Output	Asserted when the port has finished generating read command, signaling that a new read operation can be strobed. Note that while all read commands have been sent, they may not have been processed yet (which is what the read_done_o output is used to indicate).
read_done_o	Output	Asserted when read operation is completed, all commands have been applied to the DRAM controller and all data has been output.
read_data_o	Output	Data port (256 bits)
read_firstdata_o	Output	Asserted during the first data word output (read_data_o) of a read operation
read_lastdata_o	Output	Asserted during the last data word output (read_data_o) of a read operation
read_wr_o	Output	Output valid signal for read_data_o
read_afull_i	Input	Almost full flag, assert to throttle the data output of the port (see also READ_AFULL_DEPTH)
write_reset_i	Input	Reset signal
write_strobe_i	Input	Assert to strobe address information for write operation
write_first_addr_i	Input	First address for write operation (32 bits)
write_last_addr_i	Input	Last address for write operation (wraps on this address) (32 bits)
write_done_o	Output	Asserted when write operation is done
write_data_i	Input	Data port (256 bits)

Signal name	Direction	Description
write_last_i	Input	Stops the write operation (assert synchronously with the last data word to be written). If this is not asserted, the write operation will wrap over the first/last address space.
write_empty_i	Input	Empty signal, assert to stop the port from reading data
write_read_o	Output	Read signal, write_data_i will be captured and written when this is asserted

5.2.1 Parameter READ_AFULL_DEPTH

Each reader port is instantiated in multiport top with a parameter called READ_AFULL_DEPTH. The data chain for reading from DRAM looks like below, if we simplify away the other ports:

The data reader port will send out bursts of read commands into the command FIFO, and will not stop until the module which is connected to the read port sends its ”almost full” signal. However, since the command FIFO can contain several commands, this means that even though the read port stops sending more commands when the ”almost full” is received, there will still be some extra writes done depending on how many commands were already in the writer FIFO when the full signal was received. There is also a FIFO and pipelining in the DRAM + DRAM controller which can hold some pending commands

The READ_AFULL_DEPTH sets how many read commands the port is allowed to have in the writer FIFO and DRAM loop at any given time, before pausing and waiting for data to come back. This parameter should therefore be set to less than the remaining amount of rows in the receiving module FIFO when almost full is sent

On ADQ7 READ_AFULL_DEPTH is 128. It is recommended to add at least 8 to this for the almost full limit to account for delay in the DRAM controller.

5.2.2 Other useful DRAM info

The DRAM chips contain a number of banks. Each bank has a number of rows, which in turn has a number of columns. When data is to be accessed, the desired row is first cached in a row register (in the chip), and the desired column is then read out to the DRAM controller.

Whenever a new row is accessed, the old row must be written back to memory, and the new one read out. This is called a row switch, and is fairly costly in terms of latency.

5.3 Using GPIO

The ADQ7 in PCIe and PXIe form factor has a GPIO connector (see Fig. 2) that are connected to signals in User Logic 2. There are 12 bidirectional single-ended signals, three differential LVDS outputs and four differential LVDS inputs.

The gpio_dir_o signal controls both the tristate buffers on the FPGA and the drivers on the board. Note that each direction bit sets the direction for two GPIO bits. Bit 0 controls GPIO0 and GPIO1, bit 1 controls GPIO2 and GPIO3 and so on.

Table 9: GPIO signals in User logic 2

To/From physical pins	To/From CPU	Description
gpio_in_i	gpio_in_o	Input
gpio_out_o	gpio_out_i	Output
gpio_dir_o	gpio_dir_i	Direction signal. 1: Input, 0: Output
gpdi_in_i	gpdi_in_o	Differential input
gpdo_out_i	gpdo_out_o	Differential output

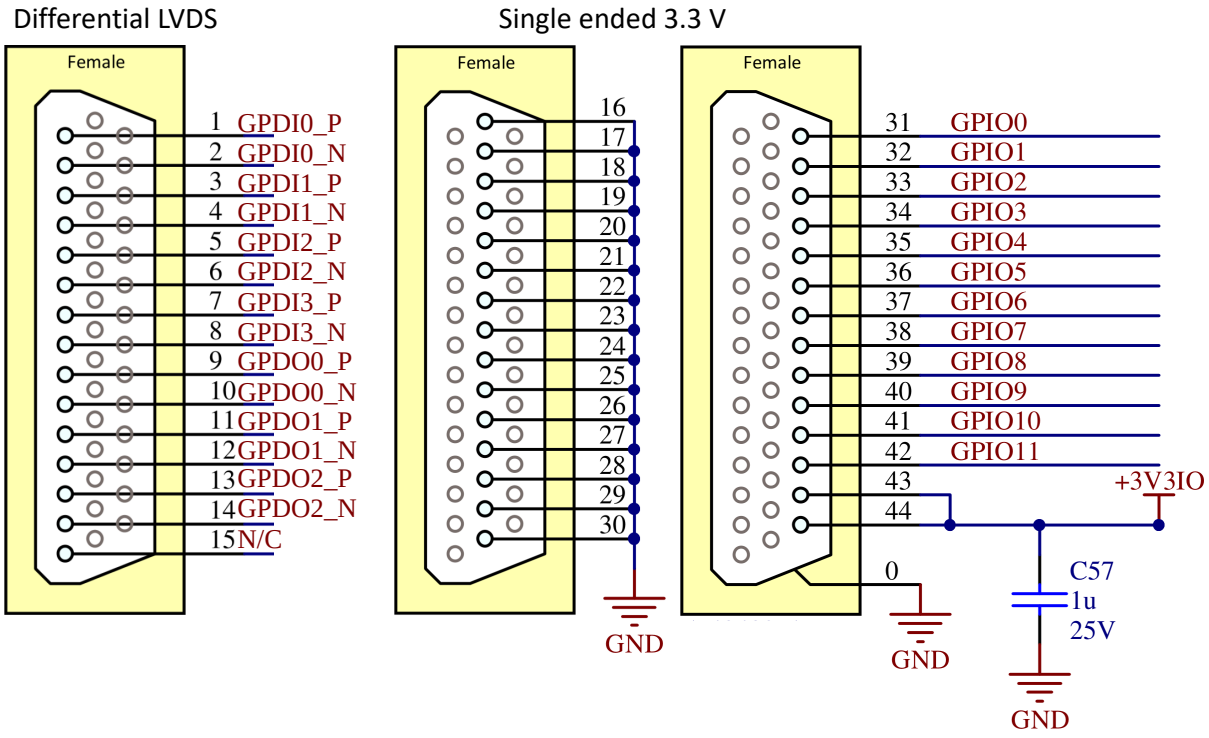


Figure 2: The GPIO connector. The voltage level of single ended GPIO pins is 3.3 V. The 3.3 V supply allows for a maximum current of 0.5 A.

5.4 Handling the record bits (User Logic 2 Only)

The record bits are used to signal the beginning and the end of a record. These bits are only used for triggered streaming. Fig. 3 presents a timing diagram of the record bits. For raw streaming, the `data_valid` signal is used to control the data The data valid signal is used to indicate that the data

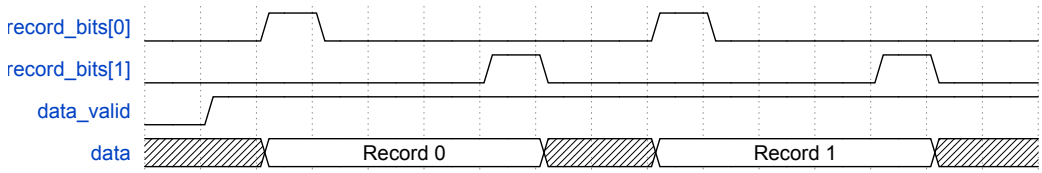


Figure 3: Timing diagram for the record bits.

during the current clock cycle is valid ADC data. The record bits are a 2-bit vector that can be extracted from the bus.

- The first bit of the record_bits_in vector indicates the start of the record. The cycle where the first bit is asserted is also part of the record.
- The second bit of the record_bits_in vector indicates the end of the record. The cycle where the second bit is asserted is also part of the record.

5.4.1 Extracting the record bits

The record bits can be extracted from the bus with the bus macro

```
extract_record_bits(ch)
```

where X is the channel. For example to extract the record bits from channel A

```
wire [1:0] record_bits_in;
record_bits_in = extract_record_bits(CH_A);
```

For example, a signal, ‘valid_record‘, which is 1 during a record and 0 otherwise, may be created by

```
reg valid_record_reg;
wire valid_record;
always @(posedge CLK_SIGNAL) begin
  if(record_bits_in[1] & data_valid_in) begin
    valid_record_reg <= 1'b0;
  end
  else if(record_bits_in[0] & data_valid_in) begin
    valid_record_reg <= 1'b1;
  end
end
assign valid_record = (record_bits_in[0] & data_valid_in) | valid_record_reg;
```

5.4.2 Inserting the record bits

The record bits are used when transferring the data to the host. Therefore, the bits have to be present on the bus after the user logic block. This can either be solved by utilizing the ‘BUS_PIPELINE’ variable, or by inserting the record bits on the bus manually.

Using the BUS_PIPELINE variable

If the length of the output data from user logic is the same as the input length, it’s recommended to use the ‘BUS_PIPELINE’ variable.

For example, if the record length is 1024 samples, and the latency of the custom logic is 10 cycles. Setting ‘BUS_PIPELINE=10’ will delay the bus (and the record bits) by 10 cycles. This will ensure that the record is framed properly by the record bits after the user logic block.

Note

The ‘BUS_PIPELINE’ delay is not applied to signals inserted through the `insert_` macros.

Inserting the record bits manually

If the length of the output differs from the input, only delaying the record bits will no longer frame the data correctly, therefore the bits have to be inserted on the bus as well.

For example, if the input data is 1024 samples, and the output data is 128 samples the record bits have to be generated to frame the new data. The ‘record_bits_in[0]’ has to be asserted for the first cycle of the record, and ‘record_bits_in[1]’ has to be asserted for the last cycle of the record. The data valid signal must also be asserted when any of the record bits are asserted. The record bits are inserted with the bus macro

```
insert_record_bits(record_bits_out, X)
```

where X is the channel. For example, to insert the record bits for channel A

```
wire [1:0] record_bits_out;
// ...
insert_record_bits(record_bits_out, CH_A);
```

Even when the record bits are inserted manually, it's important to set the 'BUS_PIPELINE' variable correctly. The other header information, e.g. the timestamp and trigger vector are sampled on the record start bit. If the latency of the user logic block is unknown, the other header fields have to be extracted and inserted as well. If the other header fields aren't of interest this can be ignored. Data will still be received, however the header information will be invalid.

5.5 Using MLVDS in MTCA backplane from User logic 1

The control of MLVDS in the MTCA backplane is given to the user, using ports `mlvds_rx_i`, `mlvds_tx_i`, `mlvds_rx_o` and `mlvds_tx_o`. The default user logic code only relays the information from the trigger module. The direction is set by SetDirectionMLVDS API, and all 8 MLVDS can either be configured as input (default) or output, individually.

The information from the trigger module to the MLVDS outputs are using ports `mlvds_rx_o_from_datatrig_i[3:0]` and `mlvds_tx_o_from_datatrig_i[3:0]`.

Note

These can also be used to transport the configured output trigger information back to User Logic 1, regardless of using the actual MLVDS or not.

The mapping of ports on User logic 1 vs the MLVDS are given by the following table:

Table 10: MTCA MLVDS mapping to User logic 1

To/From physical pins	To/From User Logic	Direction
RX17	`mlvds_rx_i[0]`	Input
RX17	`mlvds_rx_o[0]`	Output
RX18	`mlvds_rx_i[1]`	Input
RX18	`mlvds_rx_o[1]`	Output
RX19	`mlvds_rx_i[2]`	Input
RX19	`mlvds_rx_o[2]`	Output
RX20	`mlvds_rx_i[3]`	Input
RX20	`mlvds_rx_o[3]`	Output
TX17	`mlvds_tx_i[0]`	Input
TX17	`mlvds_tx_o[0]`	Output
TX18	`mlvds_tx_i[1]`	Input
TX18	`mlvds_tx_o[1]`	Output
TX19	`mlvds_tx_i[2]`	Input
TX19	`mlvds_tx_o[2]`	Output
TX20	`mlvds_tx_i[3]`	Input
TX20	`mlvds_tx_o[3]`	Output

6 Debugging on real hardware with Vivado Debug Core

There are different ways to add a Debug Core to the Vivado project. The procedure described in here is just one of the many possible ways that can be used to debug your custom logic that has been implemented with the ADQ Devkit on the real ADQ7 hardware. When inserting your Verilog code into the user_logic, you can mark the signals which you wish to probe by using the debug macro. For example:

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

Once you are done with your custom code insertion, you can run the Tcl command

```
devkit_synth_ul 1
```

and

```
devkit_synth_ul 2
```

This will create the netlists for both user_logic modules. After creating the netlist for both user_logic modules, click on

```
Run Synthesis
```

and wait for Vivado to finish synthesizing the whole design. Once it's done, click on

```
Open Synthesized Design
```

Then run the Tcl command.

```
refresh_design
```

then click on

```
Setup Debug
```

and follow the dialog screen to select the signals to probe. At this stage, you should be presented with the signals that you already marked for debug in your source code. Once you have confirmed your debug signals and the Debug Core generation is completed, click on

```
Generate bitstream
```

on the left hand panel. You will be prompted to save the constrain file with the changes made by inserting the debug core. Click yes and continue. Vivado will start generating a *.bit file containing the debug core. When this process has finished, run the Tcl command

```
devkit_mcs
```

to convert the `*.bit` file into an `*.mcs` file. Open the folder

```
implementation/DevKit.runs/impl_1
```

and find the file named

```
debug_nets.ltx
```

Save this file to another location to use with Xilinx Hardware Manager later when connecting the ADQ to the JTAG cable.

In summary, you should have 3 files in total:

- adq7.bit
- adq7.mcs
- debug_nets.ltx

Both the *.bit file and *.mcs file are functionally copies of each other. But to avoid confusions and mistakes please only use the adq7.mcs to upload to your device. The *.mcs file is not volatile as the *.bit file and it will keep the debug core intact even after you have power-cycled your device. Using the *.bit file will require you to re-upload the *.bit file again every single time you reboot your device. There is also a risk of system failure when using the *.bit file if your ADQ is connected via a P*le interface. Since uploading the *.bit file via the JTAG will reset the FPGA abruptly, P*le connection will be lost in mid-air and cause system failure. These behaviors are not specific to ADQ Devices. They are default behaviors and are common knowledge.

Depending on which clock signal you have chosen to be the master clock for your Debug Core, you might need to initiate the FPGA to get that master clock running before Vivado Hardware Manager can find the Debug Core on the ADQ7. If you have chosen the data clock as the clock for your probe, the best way to initiate that clock is to simply start ADCapturelab once and shut it down. That should be enough to initiate the clock for the Debug Core.

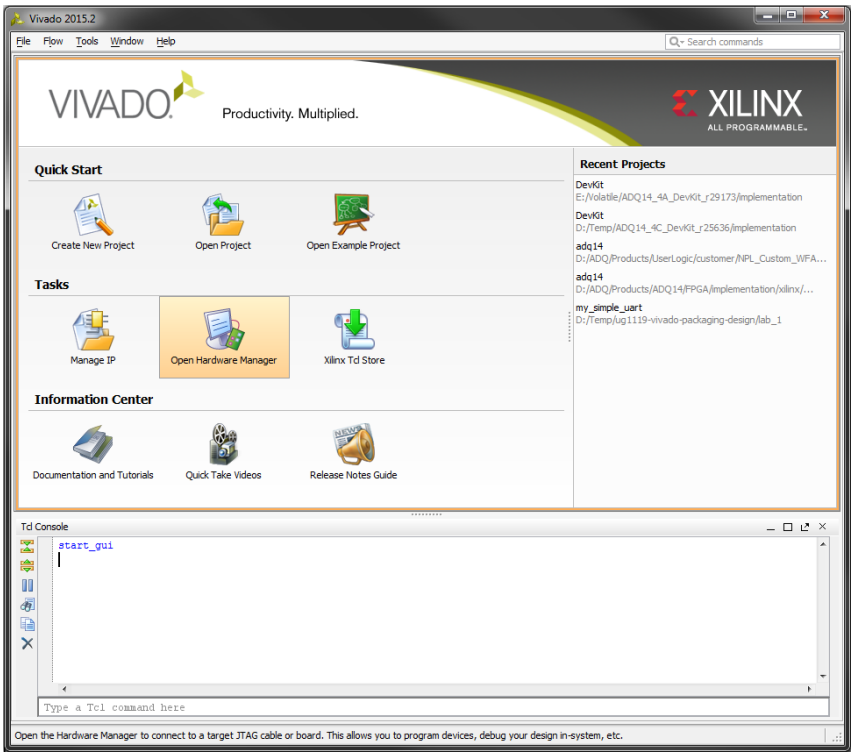
Important

The clock used for the Debug Core has to be running for the core to function.

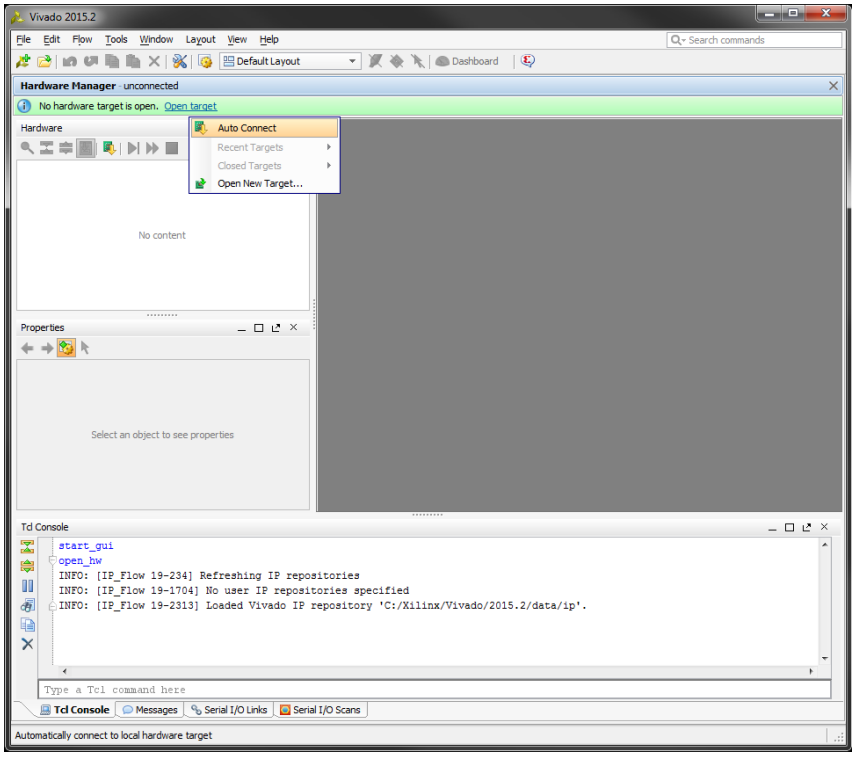
Once you have uploaded the *.mcs file containing the debug core, follow these step to start probing your debug signals.

6.1 Connecting to the debug core

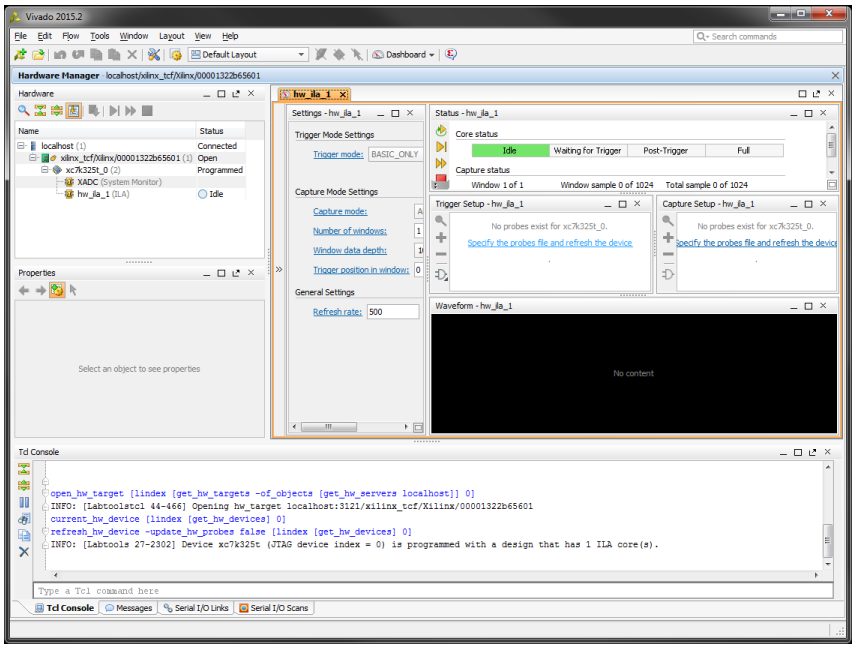
1. Start Vivado and click on Open Hardware Manager.



2. Click on Open Target and chose *Auto Connect*.

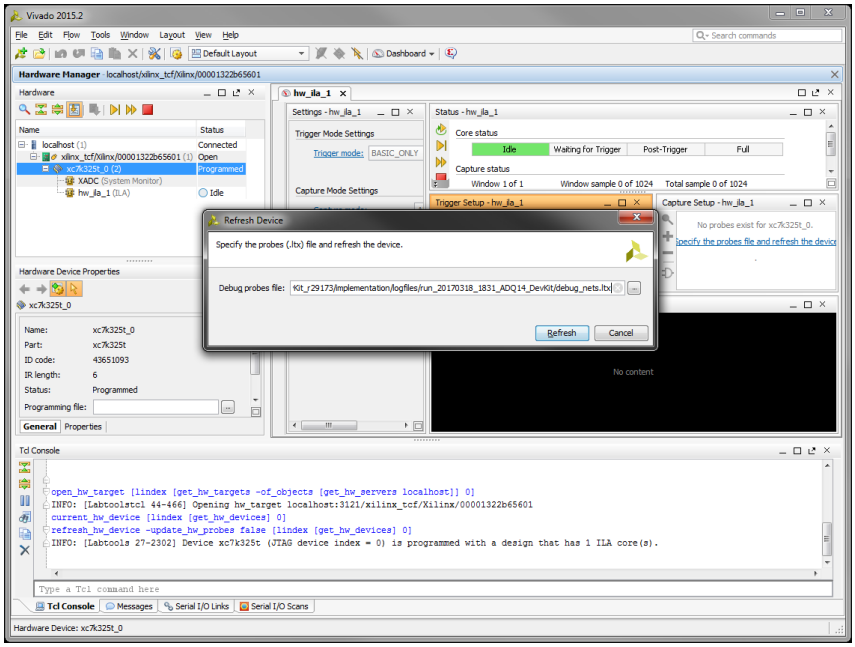


3. In the Trigger Setup window, click on *Specify probe file and refresh device*.



4. Browse to the file `debug_nets.ltx` which you have saved earlier and click on refresh. You can now

start probing your debug signals.



7 Using VHDL instead of Verilog

All code in the DevKit is Verilog. However, Vivado lets you mix Verilog and VHDL code without limitations, so you can choose to write your user code in VHDL.

Instantiate your VHDL module (see figure below – user_module_XXX.vhd) in the user logic code (user_logic1.v or user_logic2.v) in Verilog-style, and then you can write the modules in any of the languages you want. The tools will accept both Verilog and VHDL, the only important thing is that the instantiation interface is correct and same as the implemented. The simulation tools in Vivado will let you do mixed Verilog/VHDL simulation.

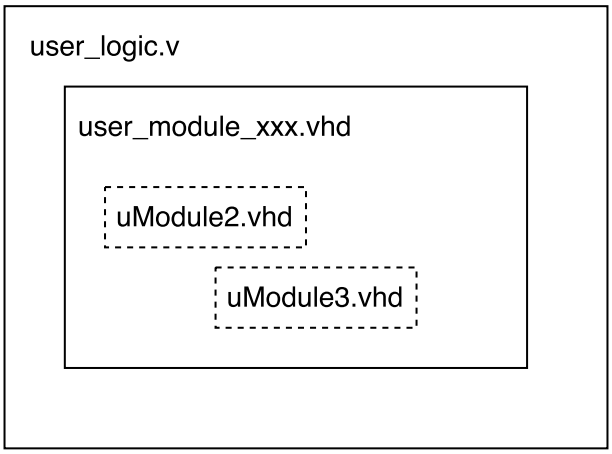


Figure 4: Hierarchy required to use VHDL

The top-level itself relies on macros (for the AXI bus extractions and insertions for instance) in Verilog, so it cannot be altered to VHDL. But after extraction and before insertion you can for instance pass on all ports (or the subset you need) to a submodule written in VHDL

TELEDYNE SP DEVICES
Everywhereyoulook™

Worldwide Sales and Technical Support
www.teledyne-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