TELEDYNE SP DEVICES
Everywhereyoulook™

ADQ8-FWDAQ Development Kit

User Guide

Author(s): Teledyne SP Devices
Document ID: 19-2290
Classification: Public
Revision: PA2
Print date: 2019-08-06

Contents

1 Introduction

This document is the user guide for the data acquisition firmware's development kit for the ADQ8 digitizer. There are different versions of the development kit depending on which device-to-host interface and number of channels of ADQ8 that is targeted. Make sure the development kit matches the target hardware.

The development kit centers around two *user logic areas*: UL1 and UL2. 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 5, 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 6, operates on complete records, potentially with a reduced sample 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
DCP	Device checkpoint—represents a saved design state in Vivado.
DevKit	Development kit
FPGA	Field-programmable gate array
FWDAQ	The Data acquisition firmware
GiB	Gibibyte (1024^3 bytes)
OOC	Out of context
PROM	Programmable read-only memory
PS	Parallel samples
`QX.Y`	Fixed-point representation with `X` integer bits and `Y` fractional bits.
RTL	Register transfer level
Tcl	Tool command language—scripting language used in Vivado.
UL1	User logic 1—the first open FPGA area, see Section 5.
UL2	User logic 2—the second open FPGA area, see Section 6.
VHSIC	Very high speed integrated circuit
VHDL	VHSIC hardware description language
Verilog	Hardware description language
Vivado	Xilinx FPGA design suite
XCI	Xilinx core instance

2 Prerequisites

The development kit has the following prerequisites:

- A license for the ADQ8 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 2018.2* is required.
  - Minimum tooling is the Vivado *Design Edition*.
  - The Vivado *WebPack* does not support the family of ADQ products.
  - 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.

Warning

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

Important

The archive should be extracted to a directory where the user has read and write permissions.

The archive is organized as follows:

`<Archive root>/`

`constraints/`	Contains the constraint files for the design.
`documentation/`	Contains the documentation for the development kit.
`edif/`	Contains the device checkpoint (DCP) file of the surrounding FPGA design.
`elf/`	Contains the Microblaze PROM file.
`implementation`	Contains the Tcl scripts that abstracts several workflow operations.
`ip/`	Contains the configuration files for the Xilinx IPs used in the design.
`source/`	Contains the Verilog source files for modules used in the design.
`License.txt`	Development kit license file.

Note

Though every file in the `source/` directory is available for editing, only a few files should be edited. This is explained further in Sections 5 and 6 which deals with the two user logic areas.

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 `<Archive root>/implementation/scripts/devkit.tcl`. The Tcl console will output the following text:

```
** ADQ8 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_update_filesets  - Update user logic filesets
  devkit_mcs              - Generate .mcs firmware file
```

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 ADQ8 works a bit diffrenty compared to ADQ7 and ADQ14 since it uses out-of-context runs for the user logic builds. For details see Section 3.5.

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 few moments to finish since parts of the design will need to be compiled. 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.

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 `implementation/` 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 firmware on the ADQ8 digitizer.

3.5 Working with the Design

This section describes the concepts of the development kit and gives an introduction to the workflow of adding customized logic functions to the digitizer firmware.

3.5.1 User Logic Runs

Each user logic area has an assosiated out-of-context (OOC) run in the Vivado project, as illustrated in Fig. 1. If there has been changes in a run's associated files it will automatically be started when building the design. If there are no changes cached results will be used when building the project. The runs can also be launched manually either by using the GUI or running the command

```
devkit_synth_ul 1
```

or

```
devkit_synth_ul 2
```

in the Tcl commad window.

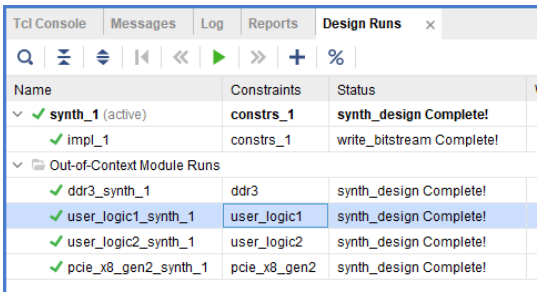


Figure 1: View of the *Design Runs* tab showing the runs for the user logic areas.

3.5.2 Filesets

When adding RTL or IP files to the project, they are included in the fileset `sources_1` by default. Since UL1 and UL2 are built out-of-context, files must be moved into the fileset for these runs. Otherwise, they will not be found when synthesizing the design. Fig. 2a illustrates a file that has just been added to the project. The module in this file is instantiated in `user_logic2.v` and therefore it must be moved to its fileset. Moving files can in most cases be done automatically by running the command

```
devkit_update_filesets
```

in the Tcl console. The move operation can also be done manually with

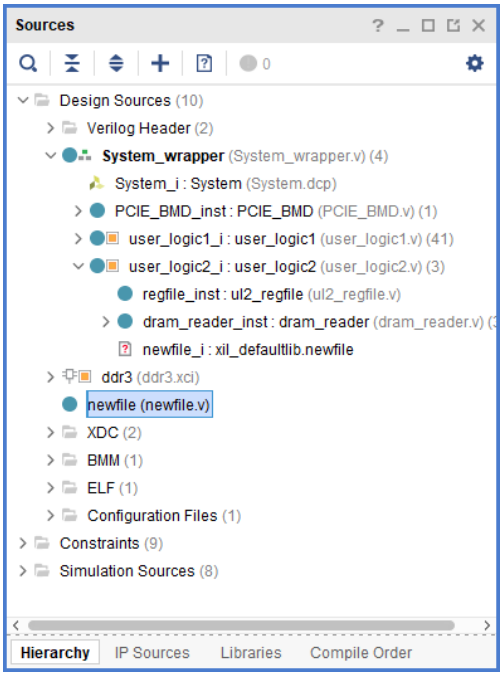
```
move_files -fileset user_logic2 [get_files newfile.v]
```

Fig. 2b shows the source hierarchy after the file has been moved.

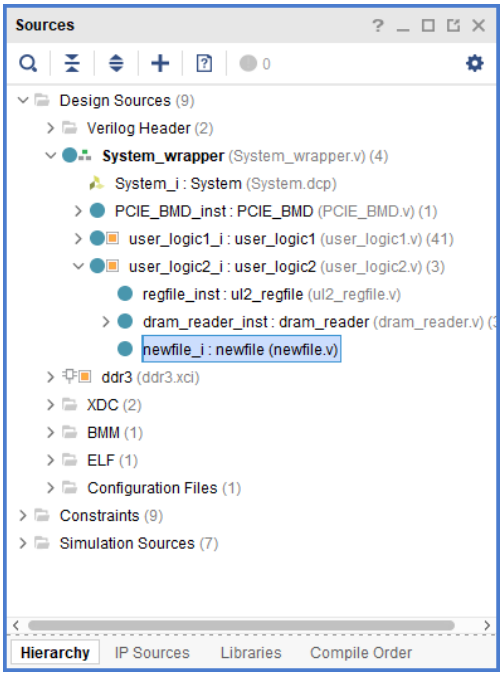
3.5.3 Typical Design Flow

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

1. Set up the development kit project as described in Section 3.3.



(a) The file `newfile.v` has just been added but UL2 cannot find it.



(b) After the move operation `newfile.v` is in the fileset for UL2.

Figure 2: Illustration of the source view in Vivado before and after a file has been moved to the fileset for user logic 2.

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 relevant *bus signals* using the *bus extraction macros* (see Section 4.5.2).
   (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 4.5.2).
   (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) If HDL or IP files have been added execute the command

```
devkit_update_files
```

in the Tcl console.

(b) Start builds by selecting *Generate Bitstream*. This action will rebuild user logic runs that are out of date and end with the bitstream generation.

(c) 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 `implementation/` directory after the process is complete.

4. 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.

5. Test the custom firmware using either

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

4 General Concepts

This section introduces concepts surrounding the development kit for ADQ8 in general and ADQ8-FWDAQ in particular. The reader is assumed to be familiar with digital design.

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, due to the many pitfalls inherent to the former. Instructing the reader on parallel design is outside the scope of this document but moving forward, some familiarity with the concept is expected.

In ADQ8-8C, the FPGA is clocked at 250 MHz while the data rate is 1 GSPS. Since a data word is equal to a sample, a parallelization of 4 is required.

4.2 Data Flow

Fig. 3 presents an overview of the data path of ADQ8-FWDAQ where the two user logic areas are highlighted. A brief description of each block in the data path is provided. Throughout this section, knowledge of concepts surrounding parallel digital design is assumed. A short summary is presented in Section 4.1. A *cycle* may be used to refer to a *data clock cycle*.

The data propagates between the modules in the data path using an *AXI stream* bus interface with custom insertion and extraction macros for convenience (see Section 4.5.2). Additionally, each module has access to the control bus via an *AXI* bus interface. 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 module

The trigger module is tasked with decoding and inserting *trigger events*, as well as the ADC data, on the data bus. A trigger event consists of a single bit indicating the event itself and additional information such as the position of the event within the data word. Additionally, functions involving the timestamp are also located in this module.

User logic 1

The first user logic area in the design. Refer to Section 5 for a more detailed description of the module.

Sample skip

The data rate can be reduced by the sample skip function. Setting the sample skip factor to, for

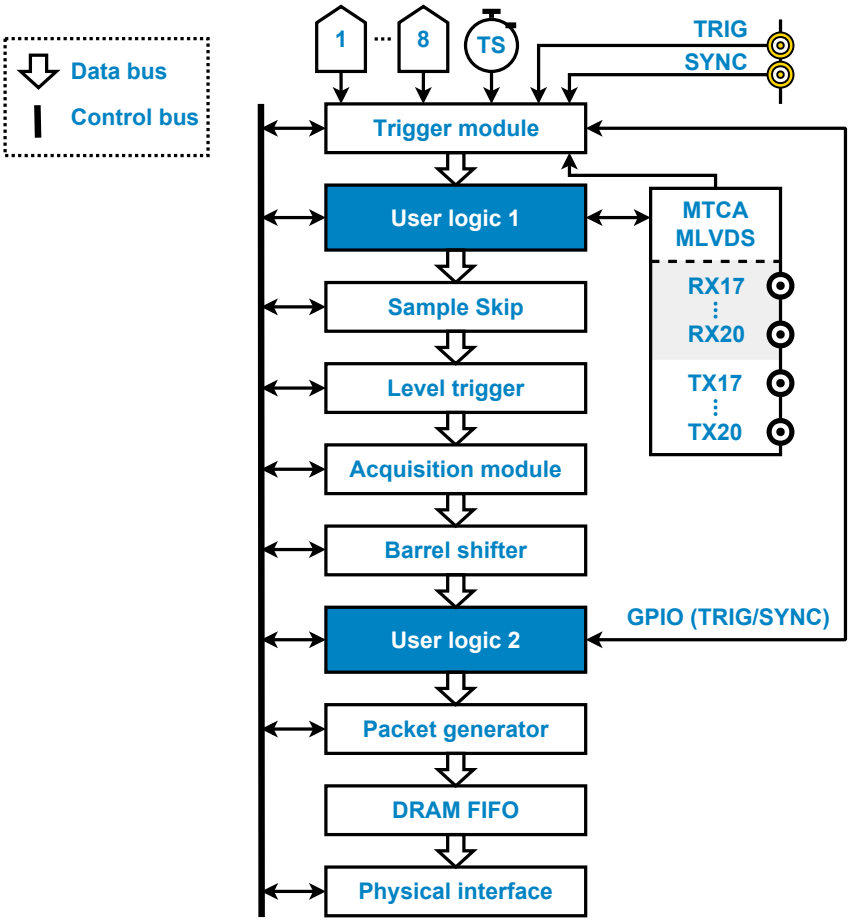


Figure 3: A block diagram of the data path of ADQ8-FWDAQ. The two user logic areas are highlighted.

example, 4 means that every 4th sample is kept and the others are discarded. Refer to the ADQ8 manual [2] for details on the settings available for this module.

Level trigger

This module is used to generate trigger information based on the amplitude of the data within the data stream. Refer to the ADQ8 manual [2] for additional details on the various settings available for this module.

Acquisition module

The acquisition module is tasked with framing the channel data into *records*. A record consists of a continuous number of samples where the starting point and stopping point is marked by a logic-high pulse on the record start and record stop data bus signals, respectively. Since records can only begin and end once per data clock cycle, the minimum record length is one data clock cycle and the record length must be divisible by the degree of parallelization.

Barrel shifter

This module rearranges the data so that the first parallel word is the first sample in the record.

User logic 2

The second user logic area in the design. Refer to Section 6 for a more detailed description of the module.

Packet generator

The packet generator is tasked with converting the information on the data bus into packets. While this operation adds some overhead to the data stream, the benefits outweigh the drawbacks since the data is more manageable in packet form when transmitting over a physical interface.

DRAM FIFO

The packet generator is followed by a DRAM FIFO with a capacity of 1 GiB. This scheme allows the digitizer to buffer data in the event of a temporary stall on the physical interface. When data is discarded due to an imbalance between the acquisition rate and the readout rate, packets are discarded at the FIFO Input,

Physical interface

The digitizer's physical interface, e.g. PXIe or MTCA.

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 [3] 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 `source/` directory:

```
source/
├─ ul_cdc_sync.v          CDC synchronization module for a 1-bit signal.
└─ ul_cdc_sync_bus_ce.v   CDC synchronization of a multi-bit signal using a strobe.
```

These modules should cover any 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 5 and 6 for examples of CDC synchronization and logic making use of these register values.

4.4 Control Bus

Each user logic area has access to the control bus which provides a connection between the custom logic and the soft microprocessor located in the enclosing design. 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 microprocessor which in turn is initiated from the ADQAPI, specifically by using the functions to read or write user registers.

Access a single register

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

Access a range of registers

- `ReadBlockUserRegister()`
- `WriteBlockUserRegister()`

Note

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

The default user logic design implements a register map but the bus can also be used to interface block RAMs, FIFOs or other custom logic.

4.4.1 Control Bus Signals

Table 2 presents the signals on the control bus. An acknowledge signal *must* be transmitted as a response to all read and write requests. Otherwise, the bus will hang and the digitizer may become unresponsive.

Important

An acknowledge signal *must* be transmitted as a response to all read and write requests. Otherwise, the bus will hang and the digitizer may become unresponsive.

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.

Table 2: The signals on the control bus.

Signal	Description	Direction	Polarity
`clk`	Bus clock	Input	N/A
`rst_i`	Start-up reset	Input	Active high
`addr_i`	Read/write address, 14 bits	Input	N/A
`wr_i`	Write strobe	Input	Active high
`wr_ack_o`	Write data acknowledge	Output	Active high
`wr_data_i`	Write data 32 bits	Input	N/A
`rd_i`	Read strobe	Input	Active high
`rd_ack_o`	Read acknowledge	Output	Active high
`rd_data_o`	Read data 32 bits	Output	N/A

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.

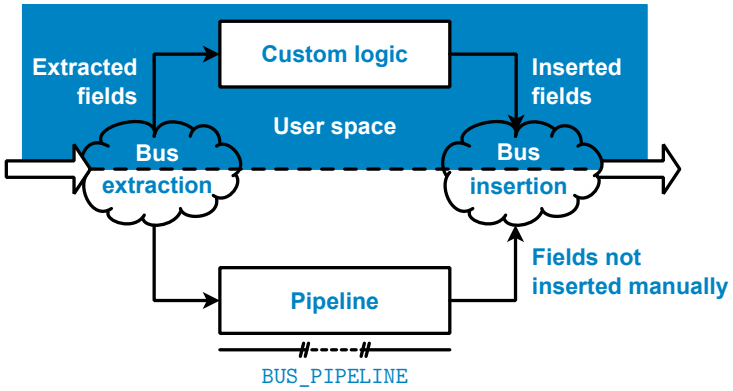


Figure 4: A diagram showing the principle of extracting signals from and inserting signals into the data bus. Any field not inserted manually is subjected to a pipeline delay equal to the value of the `BUS_PIPELINE` parameter.

4.5.1 Two Bus Definitions

Roughly halfway through the design, the data bus is redefined. At this point, a few signals are added to the bus and a few existing signals change their width. The two definitions are labeled *real-time* and *reduced rate* where the former describes the composition of the bus up until the point where it is redefined. Regardless of where the redefinition occurs, the important point is that the first user user logic area uses the real-time bus definition while the second user logic area uses the reduced rate definition. The two bus definitions each have their own set of functions to support bus operations. These are available in two Verilog header (`.vh`) files in the `source/` directory.

```
source/
├─ bus_splitter_rt.vh      Defines functions to interface with the real-time data bus.
└─ bus_splitter_rr.vh      Defines functions to interface with the reduced rate data bus.
```

4.5.2 Bus Signals

This section provides a reference for the bus signals on the real-time (RT) and the reduced rate (RR) data buses. Each section defines one bus signal, provides a description of its functionality and purpose and lists the associated bus interface functions. The interface functions are defined on the form `insert_*` and `extract_*`, where `*` is a string based on the signal name.

Section 4.5.1 explained that the definitions of these interface functions are located in two files: `bus_splitter_rt.vh` and `bus_splitter_rr.vh`. The user *must only* include the file matching the bus definition in the target user logic area, i.e.

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

Note

In the default design, the source files for the two user logic areas import the appropriate bus definition.

Timestamp *RT: single, RR: per channel*

The *timestamp* signal is tasked with providing a monotonically increasing counter to serve as a time base for the digitizer. The signal is 50 bits wide and holds an unsigned value that may be synchronized to external and internal events by using the timestamp synchronization mechanism. This feature is outlined in the ADQ8 manual [2]. The resolution of the time base on ADQ8 is 250 ps. Hence, the counter increments its value by 16 each data clock cycle.

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.

`insert_timestamp(signal)` *RT*

Insert the 50-bit timestamp signal into the real-time data bus. The timestamp `signal` is expected as an input argument.

Table 3: This table presents an overview of the signals on the real-time and the reduced rate data buses. The table is separated into two sections: one for signals common between the two buses and one for signals unique to the reduced rate data bus. Refer to the section for each individual signal for details.

`insert_timestamp(signal, channel)` *RR*

Insert the 50-bit timestamp signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestamp(DONT_CARE)` *RT*

Extract the 50-bit timestamp signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestamp(channel)` *RR*

Extract the 50-bit timestamp signal from the reduced rate data bus. The target `channel` is expected as an input argument.

Timestamp synchronization *RT: single, RR: per channel*

The *timestamp synchronization* signal is controlled by the timestamp synchronization mechanism. The signal is one bit wide and a logic high level implies that the digitizer is waiting for a timestamp synchronization event and a logic low level implies that the timestamp is in sync—provided the

level was previously logic high—or that the mechanism has not been activated.

`insert_timestampsync(signal)` *RT*

Insert the 1-bit timestamp synchronization signal into the real-time data bus. The synchronization `signal` is expected as an input argument.

`insert_timestampsync(signal, channel)` *RR*

Insert the 1-bit timestamp synchronization signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestampsync(DONT_CARE)` *RT*

Extract the 1-bit timestamp synchronization signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestampsync(channel)` *RR*

Extract the 1-bit timestamp synchronization signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Timestamp synchronization counter *RT: single, RR: per channel*

The *timestamp synchronization counter* signal is controlled by the timestamp synchronization mechanism. The signal is 16 bits wide and holds an unsigned value tasked with keeping track of the number of timestamp synchronization events since the mechanism was last armed.

The counter value during a data clock cycle where record start is asserted propagates to the user space in the host computer via the record header—provided the mechanism is set up and activated.

`insert_timestamp_sync_cnt(signal)` *RT*

Insert the 16-bit timestamp synchronization counter signal into the real-time data bus. The counter `signal` is expected as an input argument.

`insert_timestamp_sync_cnt(signal, channel)` *RR*

Insert the 16-bit timestamp synchronization counter signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_timestamp_sync_cnt(DONT_CARE)` *RT*

Extract the 16-bit timestamp synchronization counter signal from the real-time data bus. The input argument is not used by the function but must be provided nevertheless. The `DONT_CARE` parameter is defined for this purpose.

`extract_timestamp_sync_cnt(channel)` *RR*

Extract the 16-bit timestamp synchronization counter signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger event *RT: per channel, RR: per channel*

The *trigger event* is a 1-bit signal indicating that the configured trigger condition has been met in this data clock cycle. The condition is specified through the ADQAPI function `SetTriggerMode()`. The signal is active high. The bus interface is listed in Tables 4 and 5.

Trigger event edge *RT: per channel, RR: per channel*

The *trigger event edge* 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. The bus interface is listed in Tables 4 and 5.

Trigger event number *RT: per channel, RR: per channel*

The *trigger event number* is an 8-bit signal holding an unsigned value indicating where in the data word the trigger occurred. Similar to the timestamp, the signal has a 250 ps resolution but the value is aligned within the eight bits so that a `Q5.2` representation is maintained. In this representation, the value relates to the current sample rate so that

- the five MSBs form the integer part of the number—indicating, with *sample precision*, a floored value for the trigger point and
- the three LSBs form the start of the fractional part of the number, that together with the extended precision indicates, with *sub-sample* precision, the closest 25 ps grid point to the trigger point.

The value is only valid when the trigger event is asserted. The bus interface is listed in Tables 4 and 5.

Trigger extended precision *RT: N/A, RR: per channel*

The *trigger extended precision* signal is a 16-bit signal holding an unsigned value providing extended precision for the trigger point when the sample skip mechanism is active. When samples are discarded in the full-rate data stream, the effective sample rate is reduced. However, the decimal point in the trigger event number remains fixed and so does the interpretation of the field. Thus, the additional bits provided by this signal are needed to keep the high precision trigger information.

All 16 bits are interpreted as a continuation of the fractional part of the trigger event number. Concatenated, the two signals become a `Q5.18` fixed-point number indicating the trigger position within the data word, with respect to the effective sample rate. The value is only valid when the trigger event is asserted. The bus interface is listed in Tables 4 and 5.

Data *RT: per channel, RR: per channel*

The *data* signal holds the ADC data 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 target.

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 `UL1_SPD_DATAWIDTH_BITS` and the number of parallel samples as `UL1_SPD_PARALLEL_SAMPLES`. The width of a sample is always constant but the number of parallel samples differs between the two base designs (see Section 4.1).

A sample is encoded using a 10-bit *2's complement* representation.

`insert_ch_all(signal, channel)` *RT, RR*

Insert the `UL1_SPD_PARALLEL_SAMPLES` · `UL1_SPD_DATAWIDTH_BITS`-bit data signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_all(channel)` *RT, RR*

Extract the `UL1_SPD_PARALLEL_SAMPLES` · `UL1_SPD_DATAWIDTH_BITS`-bit data signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Data valid *RT: N/A, RR: per channel*

The *data valid* signal exists on a per-channel basis on the reduced rate data bus. On the real-time data bus, the signal is not present since by definition, every cycle is considered valid. The signal is one bit wide and when asserted, every sample in the data word is considered valid, regardless of the sample skip factor.

`insert_data_valid(signal, channel)` *RR*

Insert the 1-bit data valid signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_data_valid(channel)` *RR*

Extract the 1-bit data valid signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger *RT: per channel, RR: per channel*

The *trigger* is a collection of the trigger-related signals needed to support ADQ8's function set and exists on per-channel basis. Table 4 presents an overview of the trigger and its member signals.

Important

It is not recommended to extract and decode the trigger signal explicitly, but rather to use the functions targeting the individual signals within.

Table 4: This table presents the signals that together constitute the trigger for a data channel. The table is divided into two sections: one for signals common between the trigger definitions on the real-time data bus and the reduced rate data bus, and one for signals unique to the latter.

Signal	Bus interface
Common signals	
Trigger event	`insert_ch_trig_tevent(signal, channel)` `extract_ch_trig_tevent(channel)`
Trigger event edge	`insert_ch_trig_trising(signal, channel)` `extract_ch_trig_trising(channel)`
Trigger event number	`insert_ch_trig_tnum(signal, channel)` `extract_ch_trig_tnum(channel)`
Reduced rate signals	
Trigger extended precision	`insert_ch_trig_extended_precision(signal, channel)` `extract_ch_trig_extended_precision(channel)`

Overrange indicator *RT: per channel, RR: per channel*

The *overrange indicator* is a 1-bit signal indicating that *at least* one sample in the current data word has saturated to the minimum or maximum value in the range, whichever is closest.

`insert_over_range(signal, channel)` *RT, RR*

Insert the 1-bit data signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_over_range(channel)` *RT, RR*

Extract the 1-bit data signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

General purpose *RT: per channel, RR: per channel*

The *general purpose* signal is 16 bits wide and may be used to achieve various firmware-specific goals. The ten most significant bits are used by the multirecord mode on ADQ8 and must be preserved for correct opreation. The lower six bits may be used to, e.g. pass information between the two user logic areas in a well-defined manner. However, if sample skip is employed, only information passed while the record start bit 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.

Important

Make sure to preserve the 10 most sigificat bits if general purpose signal is used since it is reserved for recording of data.

`insert_ch_general_purpose_vector(signal, channel)` *RT, RR*

Insert the 16-bit general purpose signal into either data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_general_purpose_vector(channel)` *RT, RR*

Extract the 16-bit general purpose signal from either data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Record bits *RT: N/A, RR: per channel*

The *record bits* signal consists of two 1-bit signals: *record start* (bit 0) and *record stop* (bit 1). Together they frame a record. They are *inclusive*, meaning that if either signal is asserted, the data associated with that data clock cycle belongs to the record.

The data valid signal is guaranteed to be asserted if either of the record bits is asserted. It is *imperative* that any custom design in the second user logic area keeps this property.

Important

The data valid signal is guaranteed to be asserted when either of the record bits are. This property *must* be preserved when handling the bus signals in a custom design placed in the second user logic area.

Figs. 5 and 6 present a timing diagram for the record bits when sample skip is disabled and enabled, respectively. Note that record start and record stop may be asserted for multiple cycles, with only one cycle which overlaps with data valid.

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 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. 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 data valid is asserted.
- The record bits are asserted in the same data clock cycle for records which consist of one data word (the minimum record length).
- The record bits may be asserted for multiple cycles. Only one of these cycles must have data valid asserted.

- For the *multi-record* data acquisition mode, the *record length* must be preserved. For the *triggered streaming* data acquisition mode, the length may be modified. Refer to the ADQ8 manual [2] for information on the data acquisition modes.
- For the multi-record mode, the *number of records* must be preserved. For the triggered streaming mode, records may be generated or discarded in the second user logic area.

`insert_record_bits(signal, channel)` *RR*

Insert the 2-bit record bits signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_record_bits(channel)` *RR*

Extract the 2-bit record bits signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

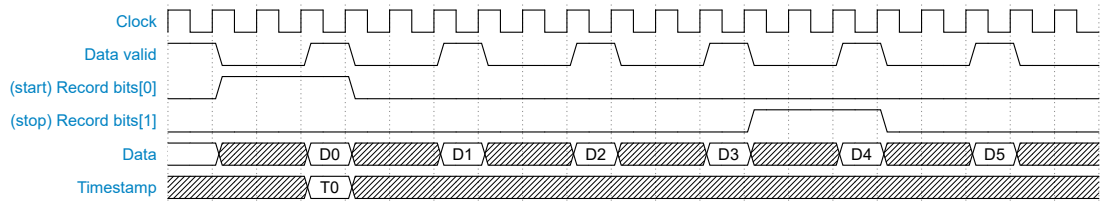


Figure 5: Timing diagram for the record bits when sample skip is disabled (skip factor 1). The figure presents two records. The first record spans ten data clock cycles and the second spans one data clock cycle.

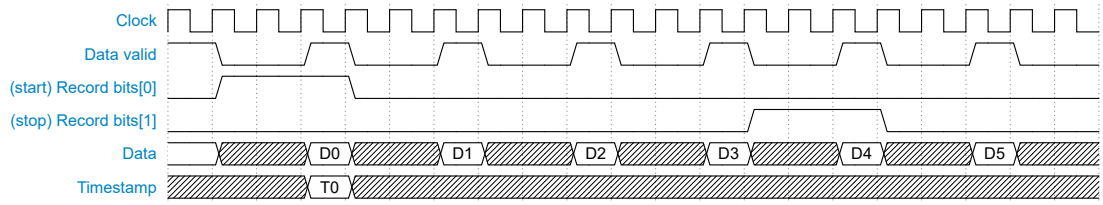


Figure 6: Timing diagram for the record bits when sample skip is enabled. The record spans five data clock cycles.

Record counter

RT: N/A, RR: per channel

The *record counter* signal holds a 16-bit unsigned value tasked with keeping track of the number of records that have been created since the digitizer was last armed. Arming the digitizer is carried out by the calling the ADQAPI functions `StartStreaming()` or `ArmTrigger()`, depending on the data collection mode. The value is only valid when record start is asserted.

`insert_record_cnt(signal, channel)` *RR*

Insert the 16-bit counter signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_record_cnt(channel)` *RR*

Extract the 16-bit counter signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

User ID

RT: N/A, RR: per channel

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 the record start bit is asserted will propagate to the user space in the host computer via the record header.

`insert_user_id(signal, channel)` *RR*

Insert the 8-bit user ID signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_user_id(channel)` *RR*

Extract the 8-bit user ID signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Trigger event bit vector

RT: N/A, RR: per channel

The *trigger event bit vector* is a signal of width equal to the number of parallel samples in the data word. Each bit in the signal is associated with the sample with the same *index* and indicates

- the *presence* of a trigger event with a logic high level and
- the *absence* of the trigger event with a logic low level.

This signal is *only* connected to the level trigger. Hence, it is only active when the level trigger is configured and enabled. This holds true regardless of whether or not the level trigger is used for data acquisition.

The signal is used to distinguish multiple events in one data clock cycle. Due to the nature of the level trigger, only *every other* sample may indicate a trigger event in the worst case.

Example

If bit 0 is asserted, the trigger condition is fulfilled for the sample at index 0 in the data word.

`insert_ch_trig_tevent_bitvect(signal, channel)` *RR*

Insert the trigger event bit vector signal into the reduced rate data bus. The `signal` and the target `channel` are expected as input arguments. The channel is indexed from zero and upwards.

`extract_ch_trig_tevent_bitvect(channel)` *RR*

Extract the trigger event bit vector signal from the reduced rate data bus. The target `channel` is expected as an input argument, indexed from zero and upwards.

Auxilliary trigger *RT: single, RR: single*

The *auxilliary trigger* is a collection of the same types of trigger-related signals as the channel trigger but operates independently. Additionally, there is only *one* auxilliary trigger signal on the data bus, as opposed to one per channel. The auxilliary trigger is configured by calling `SetAuxTriggerMode()`.

The purpose of this additional trigger signal is to serve custom designs which require that two separate trigger sources are observed. The enclosing ADQ8 design does not use the auxilliary trigger and neither is it forwarded to the host computer. It is a signal with the specific purpose of stimulating custom logic. Table 5 presents the collection of auxilliary trigger signals and their corresponding bus interface.

Table 5: This table presents the signals that together constitute the auxilliary trigger. The table is divided into two sections: one for signals common between the trigger definitions on the real-time data bus and the reduced rate data bus, and one for signals unique to the latter.

Signal	Bus interface
Common signals	
Trigger event	`insert_aux_trig_tevent(event)` `extract_aux_trig_tevent(DONT_CARE)`
Trigger event edge	`insert_aux_trig_trising(edge)` `extract_aux_trig_trising(DONT_CARE)`
Trigger event number	`insert_aux_trig_tnum(number)` `extract_aux_trig_tnum(DONT_CARE)`
Reduced rate signals	
Trigger extended precision	`insert_aux_trig_extended_precision(precision)` `extract_aux_trig_extended_precision(DONT_CARE)`

5 User Logic 1

The first user logic module uses the real-time bus definition (Section 4.5) to describe the composition of the 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, the enclosing design expects any custom design to output valid data on each data clock cycle.

Important

The first user logic area 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 `source/user_logic1_defines.vh` defines constants to use when parametrizing a design in the first user logic area.

5.1 Linear Phase FIR Filter

By default, the first user logic area contains example code implementing an order 16 linear phase FIR filter. Fig. 7 presents a block diagram of the top-level module. The top-level design demonstrates

- how to interface with the data bus (Section 4.5),
- how to modify the register map to add new control bus registers (Section 4.4) and
- how to perform clock domain synchronization (Section 4.3) of the register values forwarded to the filter's configuration interface.

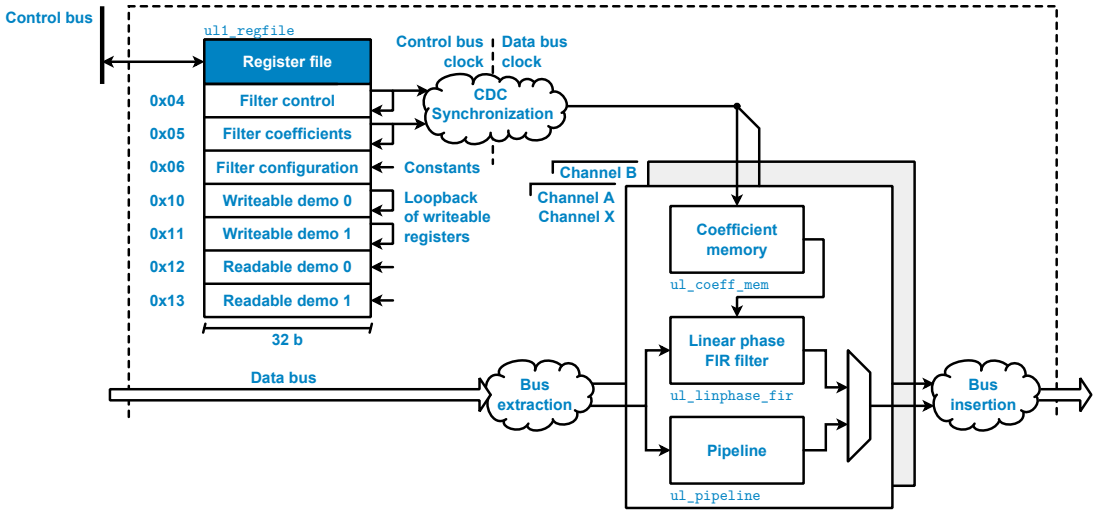


Figure 7: A block diagram of the default design present in the first user logic area. The top level contains a linear phase FIR filter of order 16.

The filter implementation is a heavily parametrized polyphase decomposed linear phase FIR filter and thus demonstrates the concept of parallel design (Section 4.1). The top level module defines parameters that may be used to change the filter properties, e.g. the filter order, coefficient width and the coefficient fixed-point representation. However, there are limitations. For example, the filter order *must* be an even number in order for the group delay to be an integer. Additionally, increasing the coefficient width will require modification of the control bus interface and its CDC structures. In most cases, the design should provide error messages upon compilation if an invalid parameter set has been provided. The filter design is separated into the following files:

```
source/
```

`ul_coeff_mem.v`	Implements the filter coefficient memory. The sychronized register values are passed to this module.
`ul_linphase_fir.v`	Implements the filter top level. The module accepts the full-width data signal as input and responds with the processed data on the same format.
`ul_linphase_fir_unit.v`	Implements the processing branch in the polyphase decomposed filter structure.
`ul_linphase_fir_bs.v`	Implements a barrel shifter to compensate for the group delay of the filter.
`ul_add_macc.v`	Implements an add, multiply and accumulate module. The internal filter structure consists of several of these modules in cascade.
`ul_saturated_rounding.v`	Implements saturated rounding for the filter output signal.
`ul_dsp_primitive.vh`	Helper file instantiating a DSP48E2 primitive when targeted by an `` `include `` statement.
`ul_clog2.vh`	Helper file defining the ceiled $\log_2$ function.

6 User Logic 2

The second user logic module uses the reduced rate bus definition (Section 4.5) to describe the composition of the 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. Since ADQ8 is only supporting multirecord data transfer the length of the record must be preserved, i.e. dynamic record length is not supported.

Note

The file `source/user_logic2_defines.vh` defines constants to use when parametrizing a design in the second user logic area.

7 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. Please include information about the use case such as the pulse detection settings as well as the specification for both the trigger and data signals. Make sure to include a trace log file from a run where the error appears.

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 7.1 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.

7.1 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* [4].

Warning

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

7.1.1 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. Generate the bitstream by clicking on *Generate bitstream*.

6. When the process has finished, run the Tcl command

```
devkit_mcs
```

7. Copy the generated files

   - `implementation/DevKit.runs/impl_1/debug_nets.ltx`
   - `implementation/adq8.mcs`
   - `implementation/adq8.bit`

   to a permanent location.

8. Program the firmware image (`.mcs` file) using ADQUpdater. Refer to the ADQUpdater user guide for instructions on how to manage the firmware on the ADQ8 digitizer [1].

7.1.2 Connecting to the Debug Core

The Vivado Hardware Manger 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 `debug_nets.ltx` file is available.

Depending on the clock signals chosen for the debug core, the firmware may have to be initialized before Vivado Hardware Manager can find the debug core. Initialization is done by calling the ADQAPI function `SetupDevice()`.

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* [4] for further instructions.

References

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

[2] Teledyne Signal Processing Devices Sweden AB, *17-2000 ADQ8 manual*. Technical Manual.

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

[4] Xilinx Inc., *Programming and Debugging*, April 2017. User Guide Guide (UG908).

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