distributed_oscilloscope Documentation

Release 0

Milosz Malczak

Aug 26, 2019

Contents:

CHAPTER 1

Introduction

The Distributed Oscilloscope (DO) is an application allowing to synchronously monitor analog signals in a distributed system, independently of the distance.

The idea of the DO is presented in Fig. 1.1.

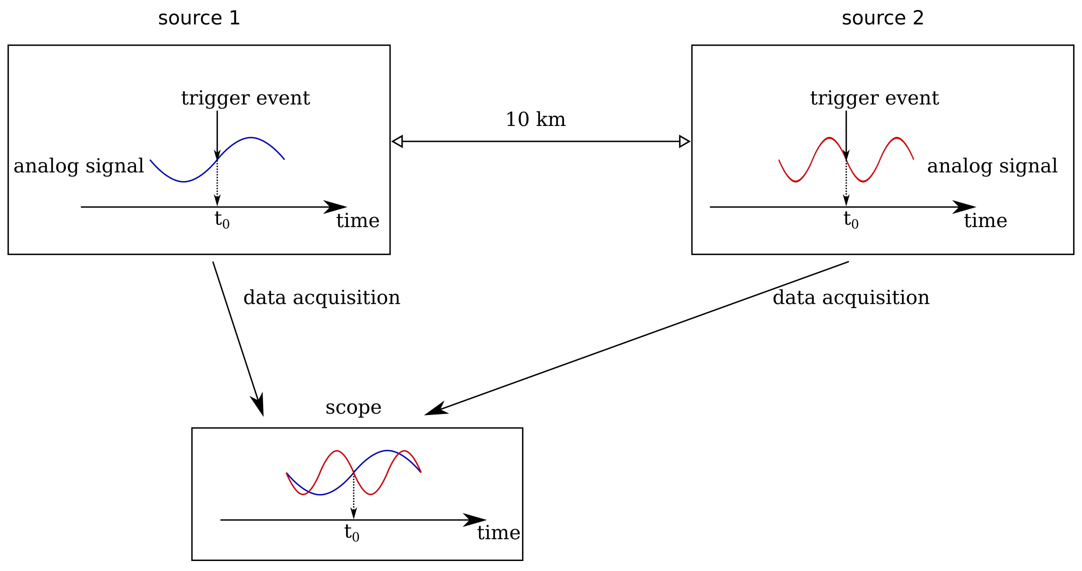


Fig. 1.1: Synchronous acquisition of distributed data

Analog signals from various digitizers are time-stamped, aligned to the same moment in time and sent to the Graphical User Interface (GUI), to be displayed. The synchronization is obtained using the White Rabbit Trigger Distribution (WRTD) project.

1.1 Architecture

The DO consists of three layers:

- *User Applications*
- *DO Server*
- *Device Application*

The structure of the DO is presented in Fig. 1.2.

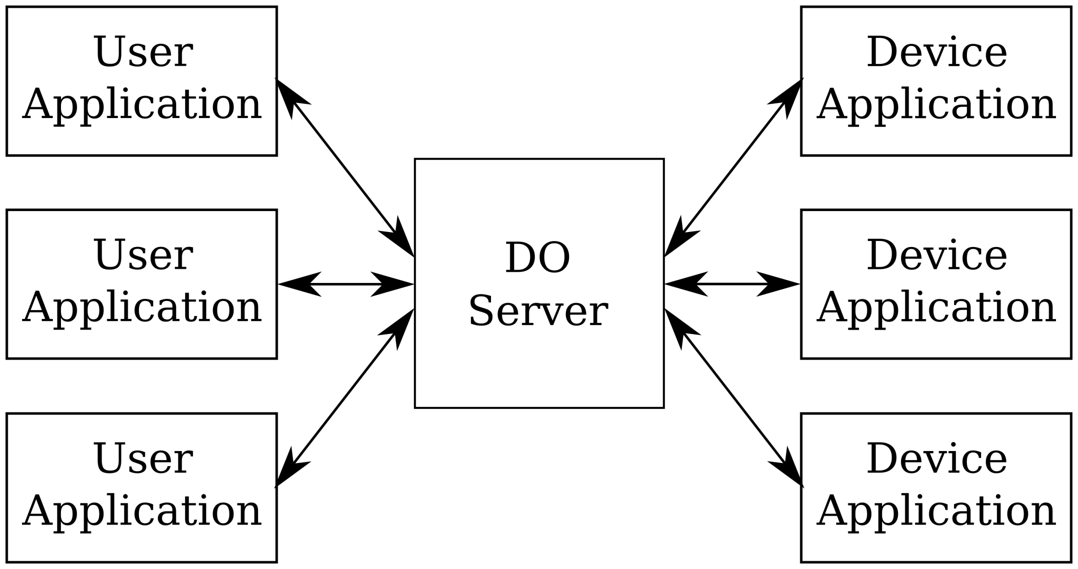


Fig. 1.2: Structure of the DO

The DO Server is a proxy between Devices and Users Applications. In single network, there could be one server, multiple users and multiple devices. The applications typically are run on different machines, but it is not a restriction.

1.1.1 User Applications

There are currently two User Applications available:

- GUI — it is designed to resemble standard oscilloscope.
- testbench — it is used to test the DO Server and the Device Applications as well as to perform statistical measurements of data acquisition speed and of the precision of the synchronization.

User Applications serve the following purposes:

- Sending the configuration settings
- Collecting and processing the acquisition data

Device Applications never communicate with the devices directly, always through the DO Server. This allows to hide all the implementation details and to provide a common interface for various types of applications. The details on how to write User Applications are described in section *Developer Guide*

1.1.2 DO Server

The DO Server is a central unit responsible for managing all the connections, preprocessing the data and providing a common interface for connected applications.

1.1.3 Device Application

Device applications provide direct access to hardware resources. At the moment the only available devices are ADCs supported by the adc-lib.

1.2 Hardware setup

The minimum hardware requirements necessary to demonstrate features of the DO are the following:

- computer with minimum 2 PCIe slots and CentOS 7.6.1810

Note: The DO is designed to run each application on a different machine. However, it is possible to run them on the same machine. To make the DO really distributed, the ADC cards should be installed in different locations in different machines. The described hardware setup should serve only as a demonstrator.

Note: CentOS 7.6.1810 guaranties that all the drivers will function properly. However, it is possible to use the DO with different OS. In case of machines where the Server and the GUI are run, the Linux version does not matter.

- White Rabbit Switch
- 2 SPEC 150T boards

  Important: The DO will work only with SPEC 150T version. Be careful not to purchase standard SPEC 45T version.

- 2 FMC ADC 100M 14b 4cha boards
- 2 fibers
- 4 SFP cages
- signal generator

The minimum hardware setup of the DO is presented in Fig. 1.3.

The SPEC boards together with ADC cards should be installed in PCIe slots of the computer and connected to any of the White Rabbit switch channels using the SFP cages and fibers. To be able to demonstrate the synchronization accuracy, the same signal from the generator should be provided to both ADCs, with cables of the same length or precisely known lengths.

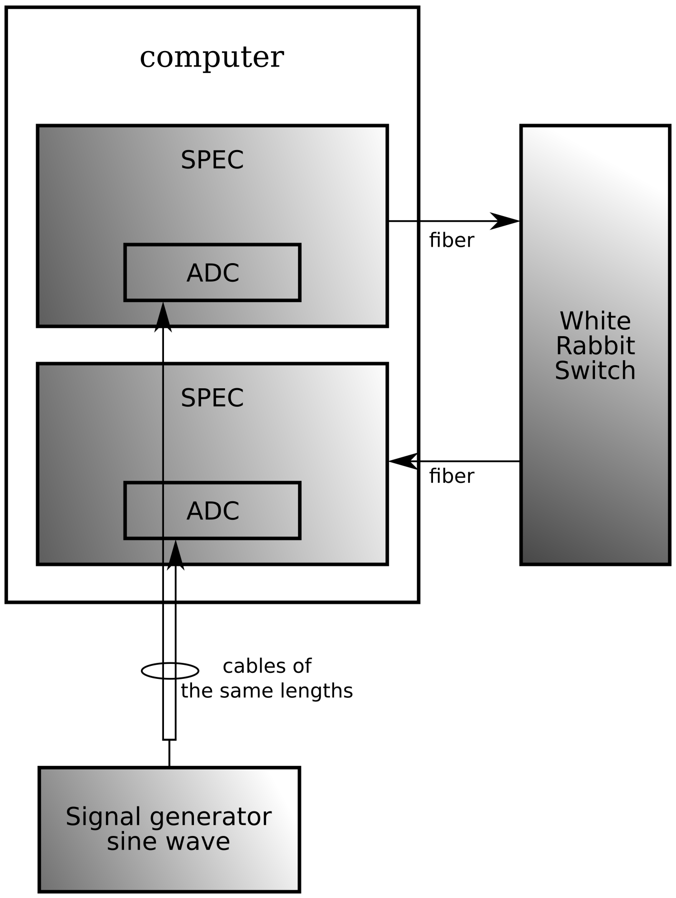


Fig. 1.3: Minimum hardware setup for the DO

CHAPTER 2

Installation

2.1 Applications

Important: To be able to access the ADC device, the *Dependencies* have to be installed.

To use the DO, the python version 3.6 is required.

Before installing the Distributed Oscilloscope and the requirements, create a Python virtual environment to avoid issues with packages versions.

```
$ python -m venv do_venv
$ source do_venv/bin/activate
```

To install the Distributed Oscilloscope, type:

```
$ pip install https://ohwr.org/project/distributed-oscilloscope/wikis/uploads/
↪96748e7016d163f85cfb146e661bdc3d/DistributedOscilloscope-1.0.0.tar.gz
```

Now, three available applications could be started form the terminal:

- dist_osc_server
- dist_osc_gui
- dist_osc_adc_node

Each of the applications requires installation of dependencies.

To install the dependencies for the Server, issue:

```
$ pip install -r https://ohwr.org/project/distributed-oscilloscope/raw/master/
↪software/DistributedOscilloscope/server/requirements.txt
```

To install the dependencies for the GUI, issue:

```
$ pip install -r https://ohwr.org/project/distributed-oscilloscope/raw/master/
↪software/DistributedOscilloscope/applications/pyqt_app/requirements.txt
```

To install the dependencies for the ADC node, issue:

```
$ pip install -r https://ohwr.org/project/distributed-oscilloscope/raw/master/
↪software/DistributedOscilloscope/nodes/adc_lib_node/requirements.txt
```

To display help for each of the applications, type the name of the applications with '-h' option, e.g.:

```
$ dist_osc_server -h
```

2.2 Dependencies

To be able to access the ADC device, the following drivers need to be loaded:

- **htvic.ko:** https://gitlab.cern.ch/cohtdrivers/coht-vic

  commit: df07c670abcf87c967b634504417e482d5e3696b
- **zio.ko, zio-buf-vmalloc.ko:** https://www.ohwr.org/project/zio/wikis/home

  commit: d8bef4d89361194c2e5644e751add9bd9ffa106d
- **fmc-adc-100m14b.ko:** https://ohwr.org/project/fmc-adc-100m14b4cha-sw/wikis/home

  commit: 54a77d73df0ef321bbe74ef4acaf2776f6a142c5
- **fmc.ko:** https://gitlab.cern.ch/fvaga/fmc

  commit: ca386f42df6cdfe5fb6462215622ab2796c2ec75
- **fpga-mgr.ko:** https://gitlab.cern.ch/fvaga/fpga-manager

  commit: a3711f798ec4a17121c2f6ccfe160fde24a170bb
- **spec.ko:** https://gitlab.cern.ch/fvaga/fmc-spec

  commit: e893e85ff45dfa3b532295b0b86c5a276b2f221c
- **mockturtle.ko:** https://ohwr.org/project/mock-turtle/wikis/home

  commit: b07df87ad36d963beb7d7596b3dffa4221d6bd58

To be able to access ADC device and WRTD, the following libraries have to be installed on the machine running the *ADC application:*:

- **adc-lib:** https://ohwr.org/project/adc-lib/wikis/home
- **WRTD:** https://www.ohwr.org/project/wrtd/wikis/home

After installing the drivers and the libraries, the SPEC150T-based FMC_ADC reference design has to be loaded. You can find the reference design here.

Todo: Reset mockturtle CPUs Enable WRTD trigger in the adc-lib

CHAPTER 3

Starting Applications

The GUI (*User Applications*) and the *DO Server* can be run on any Linux machine with python3.6. Before starting the ADC application (*Device Application*), all the dependencies, described in section *Dependencies*, have to be installed.

The first application that has to be run is the Server. When the Server is already started, GUIs and ADC nodes can be run in any order.

Before starting any of the applications, start the virtual environment and install the Distributed Oscilloscope, as described in section *Applications*.

3.1 Server Application

To start the Server Application, run in terminal:

```
$ dist_osc_server
```

Optional arguments:

- port_user – port of the Server exposed to the User Application, default value – 8003
- port_device – port of the Server exposed to the device, default value – 8023

3.2 GUI:

Before starting the GUI applications, find out what is the IP address of the Server: SERVER_IP_ADDRESS. You can check it using the command:

```
$ ifconfig
```

To start the GUI, run in terminal:

```
$ dist_osc_gui --ip_server SERVER_IP_ADDRESS
```

Required arguments:

- ip_server – IP address of the Server

Optional arguments:

- port – port used on the current machine to listen for notifications and acquisition data, default value – 8001
- port_server – port used on the Server to listen for the requests from the GUI, default value – 8003

3.3 ADC application:

If the Server and the ADC device are in different local networks, before staring the ADC applications, find out what is the IP address of the Server: SERVER_IP_ADDRESS. If the IP address of the Server is not provided, Zeroconf will be used to automatically find out this information.

Important: The Zeroconf will only work if the Server and the ADC are in the same local networks. Otherwise, the IP of the Server has to be provided manually.

To start the GUI, run in terminal:

```
$ dist_osc_adc_node --ip_server SERVER_IP_ADDRESS
```

Optional arguments:

- ip_server – IP address of the server
- port_server – port of the server used to listen for notifications and acquisition data, default value – 8023
- port – port used on the current machine to listen for the requests from the Server, default value – 8000
- pci_addr – PCI address of the desired board, default value – 0x01

3.4 Examples configuration:

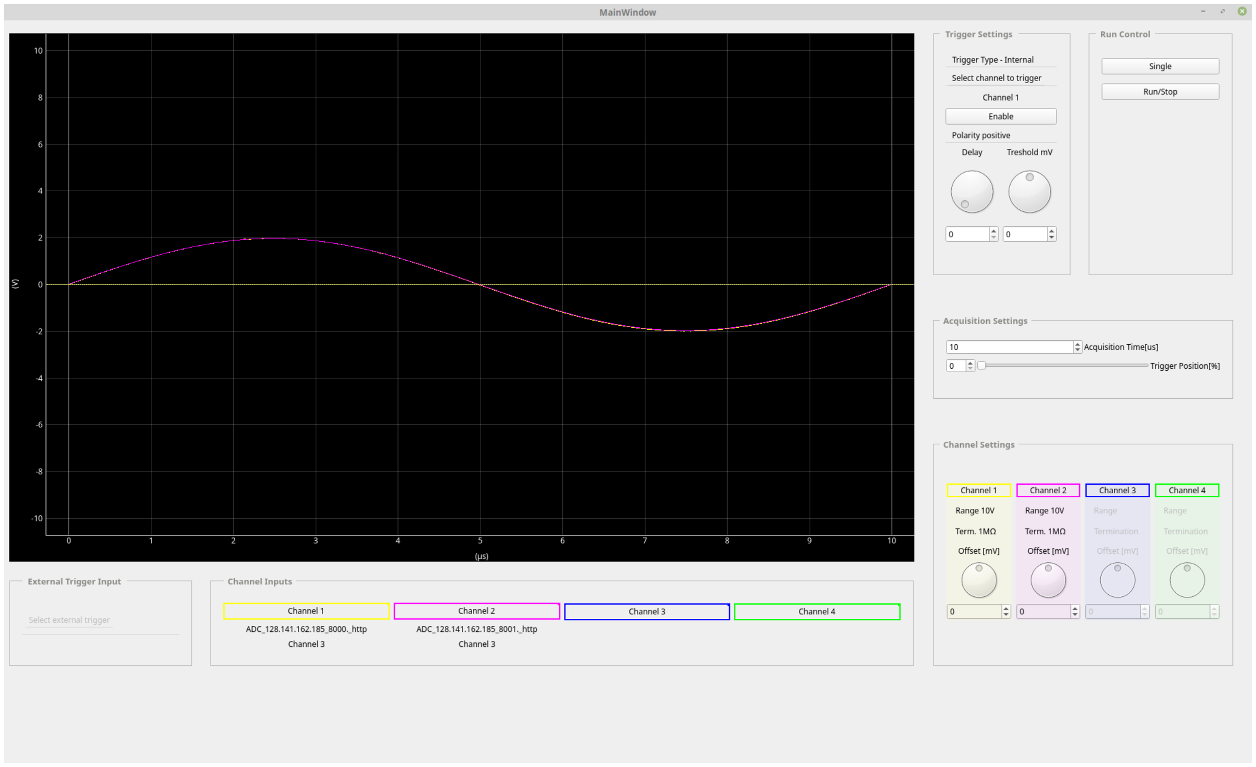
Supposing that the IP address of the Server is 128.141.79.22, the ADCs are installed in the same machine and the PCI slots where the ADCs are installed are 01 and 02, the applications have to be started with the following parameters:

```
$ dist_osc_server
$ dist_osc_gui --ip_server 128.141.79.22
$ dist_osc_adc_node --ip_server 128.141.79.22 --port 8000 --pci_addr 01
$ dist_osc_adc_node --ip_server 128.141.79.22 --port 8001 --pci_addr 02
```

CHAPTER 4

Usage of the GUI

The GUI application is presented in Fig. 4.1.



Trigger Type - Internal

Select channel to trigger

Channel 1

Polarity positive

Delay 0 Treshold mV 0

10 Acquisition Time[us]

0 Trigger Position[%]

Channel 1	Channel 2	Channel 3	Channel 4
Range 10V	Range 10V	Range	Range
Term. 1MΩ	Term. 1MΩ	Termination	Termination
Offset [mV]	Offset [mV]	Offset [mV]	Offset [mV]
0	0	0	0

Channel 1	Channel 2	Channel 3	Channel 4
ADC_128.141.162.185_8000._http	ADC_128.141.162.185_8001._http		
Channel 3	Channel 3		

Fig. 4.1: Screenshot of the GUI application

4.1 Channels selection

Just like in standard oscilloscope, there is a possibility of observing up to 4 channels. Any channel of any available ADC can be connected to the particular channel of the GUI.

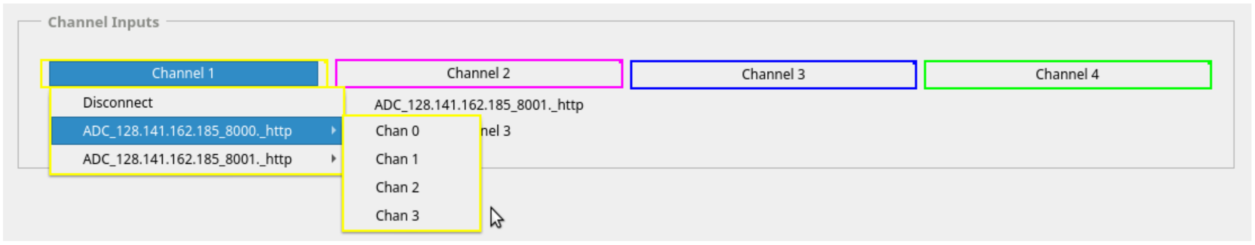

Channel Inputs
Channel 1
Channel 2
Channel 3
Channel 4
Disconnect
ADC_128.141.162.185_8000._http
ADC_128.141.162.185_8001._http
ADC_128.141.162.185_8001._http
Chan 0
Chan 1
Chan 2
Chan 3

Fig. 4.2: Selection of GUI channels

4.2 Triggers selection

The ADCs could be triggered either by external trigger pulse or when the signal of the observed channel crosses the threshold value (internal trigger).

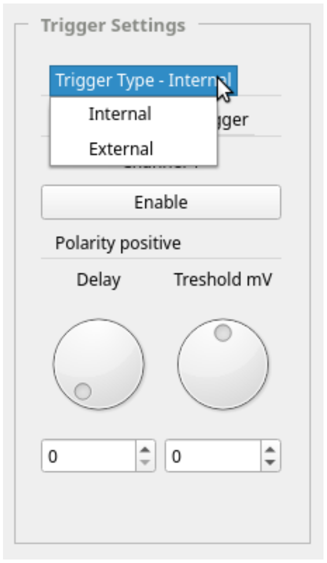


Fig. 4.3: Selection of trigger type

4.2.1 Internal trigger

If the internal trigger is selected, the GUI could be triggered on any channel to which a signal is connected.

4.2.2 External trigger

If the external trigger is selected, the GUI could be triggered by the external trigger input of any connected ADC.

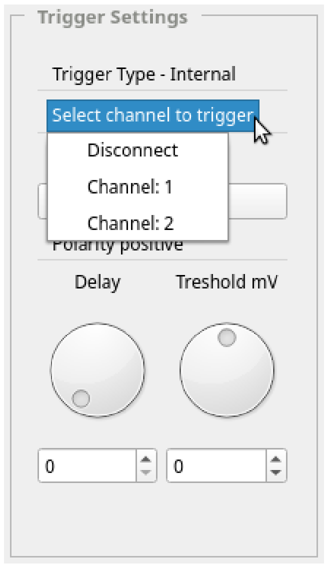


Fig. 4.4: Selection of internal trigger

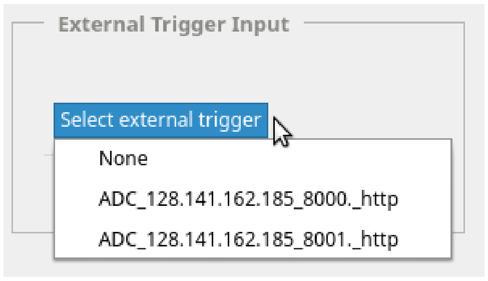

External Trigger Input
Select external trigger
None
ADC_128.141.162.185_8000._http
ADC_128.141.162.185_8001._http

Fig. 4.5: Selection of external trigger

4.3 Channels settings

Currently available channels settings are the following:

- range
- termination
- offset

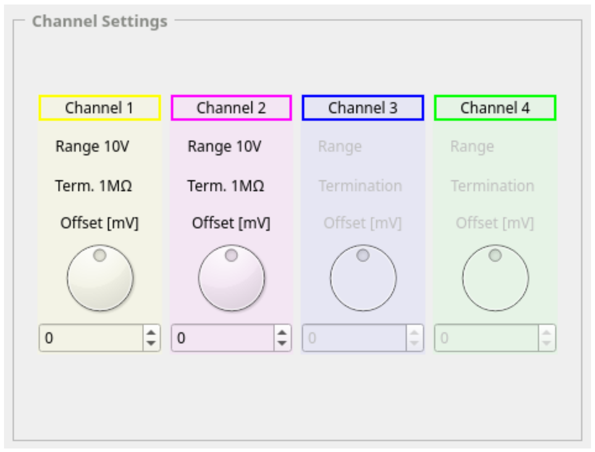


Fig. 4.6: Channels settings

4.4 Trigger settings

Currently available trigger settings are the following:

- polarity
- delay
- threshold (in case of internal trigger)

4.5 Run control

There are two available modes:

- single acquisition
- continuous acquisition

4.6 Acquisition settings

Acquisition settings allow modifying the acquisition time and position of the trigger. Position of the trigger is given in percentage of the acquisition time.

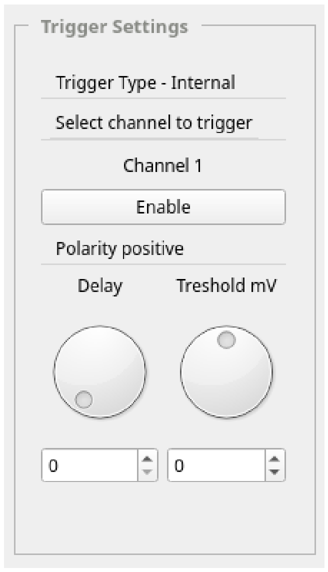


Fig. 4.7: Trigger settings

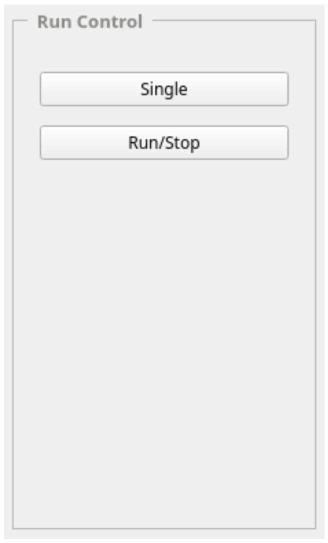


Fig. 4.8: Run control

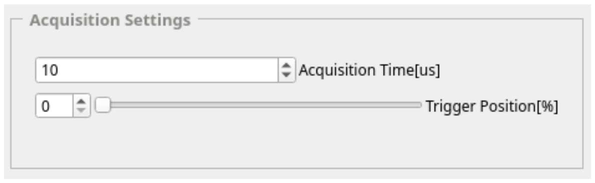


Fig. 4.9: Acquisition settings

CHAPTER 5

Developer Guide

As described in *Introduction*, DO currently supports two types of User Applications:

- GUI
- testbench

and one type of Device Application available:

- ADCs supported by the adc-lib.

Depending on the needs of the user, different applications could be developed. To do this, the following tasks have to be performed:

- write application-specific code
- update or add a new model of the application in the Server
- establish communication with the Server using the existing interface and if necessary, update the interface

The section explains briefly the communication patterns, existing interfaces and models of applications as well as the changes that have to be done to be able to add a new application.

5.1 Communication in the DO

The schematics of the communication patterns used in the Distributed Oscilloscope are presented in Fig. 5.1.

In the Distributed Oscilloscope, there are two messaging patterns used to communicate the nodes:

- request/reply pattern
- publisher/subscriber pattern

Request/reply pattern is used to implement Remote Procedure Calls (RPC), which allow controlling the behavior of other application in a reliable way. The User Applications control the behavior of the Distributed Oscilloscope, using a Server as a proxy. Therefore, the User Applications send RPC request to the Server and the Server sends the RPC requests to the Device Application.

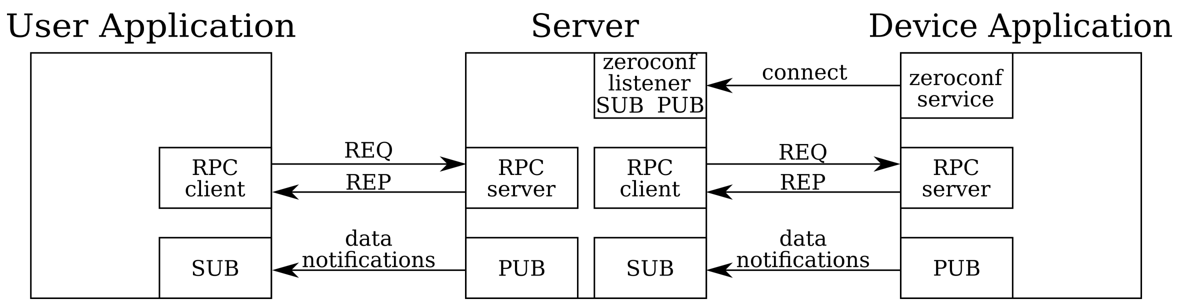


Fig. 5.1: Schematics of communications patterns in the Distributed Oscilloscope

Publisher/subscriber pattern is used for sending acquisition data and notifications about the availability of nodes. Device Applications send the notifications and acquisition data to the Server, which propagates them to User Applications.

There are two ways of providing information about the presence of the Device Application to the Server:

- if the IP address of the server is provided during the startup of the Device Application, the notification is sent over the standard communication channel – the most bottom one in Fig. 5.1.
- using Zeroconf, which automatically discovers the IP address of the Server. However, the Zeroconf Listener in the Server also uses the publisher/subscriber pattern for internal communication.

When implementing the *Interfaces* for new applications, the communication patterns should not be changed.

5.2 Applications Models

The Server contains models of User Applications and Device Applications. Interaction with the applications is done through interaction with the models, using provided *Interfaces*.

5.2.1 User Application model

The User Applications model is similar to a standard oscilloscope. The functionality of the User Applications depends on the changes made in the model, that is:

- channels selection
- triggers selection
- acquisition settings (e.g. length of acquisition, position of the trigger…)

There are no foreseen changes in the User Application model when adding a new User Application.

New applications should make use of the *Server Interface* and implement the *User Application Interface*.

5.2.2 Device Application model

The Device Application model reflects the functionalities of the particular device. In case of the ADC, the functionalities are the following:

- channels settings
- triggers settings
- acquisition settings

The main reason for adding a new Device Application is adding a new type of device. In that case, the model of the application, *Device Application Interface* and *Server Interface* should be modified.

5.3 Interfaces

The Server provides interfaces for User Applications and Devices Applications. Each new application should use these interfaces. If the interfaces don't meet the requirements for a new application, they should be modified.

5.3.1 Server Interface

Todo: Since currently the only available device is and ADC, the interface provides functions for interaction with ADCs. If in the future other types of devices will be supported, the interface should be made more general or extended.

expose.py

Exposes the functionalities of the Server to Device Applications and to User Applications. All communication with other applications is done using the Expose class. For communication with the nodes it uses ZeroMQ sockets.

class `server.expose.`**`Expose`**(*connection_manager*, *port_user*, *port_device*)
Top level class.

Parameters

- **`connection_manager`** – `ConnectionManager`
- **`port_user`** – port on which it listens for User Applications connections
- **`port_device`** – port on which it listens for devices connections

`add_channel`(*oscilloscope_channel_idx*, *unique_ADC_name*, *ADC_channel_idx*, *user_app_name*)
Called by the User Application. Adds channel in the User Application.

Parameters

- **`oscilloscope_channel_idx`** – index of the channel in the user application
- **`unique_ADC_name`** – name of the Device Application (ADC)
- **`ADC_channel_index`** – channel index of the ADC
- **`user_app_name`** – name of the User Application

`remove_channel`(*oscilloscope_channel_idx*, *user_app_name*)
Called by the User Application. Removes channel from the User Application.

Parameters

- **`oscilloscope_channel_idx`** – index of the channel in the user application
- **`user_app_name`** – name of the User Application

`add_trigger`(*type*, *unique_ADC_name*, *ADC_trigger_idx*, *user_app_name*)
Called by the User Application. Adds trigger in the User Application.

Parameters

- **`type`** – type of the trigger

- **`unique_ADC_name`** – name of the Device Application (ADC)
- **`ADC_trigger_index`** – trigger index of the ADC
- **`user_app_name`** – name of the User Application

`remove_trigger`(*user_app_name*)
Called by the User Application. Removes trigger from the User Application.

Parameters **`user_app_name`** – name of the User Application

`set_ADC_parameter`(*parameter_name*, *value*, *unique_ADC_name*, *idx=None*)
Called by the User Application. Generic function, used to modify parameters of the ADC.

Parameters

- **`parameter_name`** – name of the parameter to be modified
- **`value`** – new value of the parameter
- **`unique_ADC_name`** – name of the Device Application (ADC)
- **`idx`** – index of the given parameter if applies

`single_acquisition`(*user_app_name*)
Called by the User Application. Starts single acquisition in the given User Application.

Parameters **`user_app_name`** – name of the User Application

`run_acquisition`(*run*, *user_app_name*)
Called by the User Application. Starts or stops continuous acquisition in the given User Application.

Parameters

- **`run`** – defines if the acquisition is to be started or stopped
- **`user_app_name`** – name of the User Application

`set_pre_post_samples`(*presamples*, *postsamples*, *user_app_name*)
Called by the User Application. Defines he number of presamples and postsamples that are to be set in all ADCs used by the given User Application.

Parameters

- **`presamples`** – number of presamples
- **`postsamples`** – number of postsamples
- **`user_app_name`** – name of the User Application

`get_user_app_settings`(*user_app_name*)
Called by the User Application. Retrieves the length of the acquisition and parameters of channels and trigger used by the particular User Application.

Parameters **`user_app_name`** – name of the User Application

Returns Dictionary with required settings

`register_user_app`(*user_app_name*, *addr*, *port*)
Called by the User Application. Registers User Application in the Distributed Oscilloscope.

Parameters

- **`addr`** – IP address of the socket of the User Application
- **`port`** – port of the socket of the User Application
- **`user_app_name`** – name of the User Application

unregister_user_app(*user_app_name*)
Called by the User Application. Unregisters User Application in the Distributed Oscilloscope.

Parameters **user_app_name** – name of the User Application

update_data(*timestamp*, *pre_post*, *data*, *unique_ADC_name*)
Called by the Device Application.

Adds the acquisition data to the acquisition data queue in the ADC. Every time the new data arrives, the ADC notifies the User Application class, which checks if all rrequired data has arrived and is properly aligned. If yes, it sends the data to the actuall User Application.

Parameters

- **timestamp** – timestamp with the information about the time of the trigger
- **pre_post** – number of acquired presamples and postsamples
- **data** – dictionary with ADC channels indexes as keys, containing acquisition data
- **unique_ADC_name** – name of the Device Application (ADC)

register_ADC(*unique_ADC_name*, *addr*, *port*)
Called by the Device Application. Registers Device Application (ADC) in the Distributed Oscilloscope.

Parameters

- **unique_ADC_name** – name of the Device Application (ADC)
- **addr** – IP address of the socket of the Device Application (ADC)
- **port** – port of the socket of the Device Application (ADC)

unregister_ADC(*unique_ADC_name*)
Called by the Device Application. Unregisters ADC in the Distributed Oscilloscope.

Parameters **unique_ADC_name** – name of the Device Application (ADC)

run()
Called when the object of the class is created. It listens for messages from Device Applications (socket_ADC_listener), User Applications (socket_user_listener) and from Zeroconf (zeroconf_listener) in the loop. The monitor socket is used to monitor the state of ZeroMQ connection.

The message contains the name of the method to call. Since communication with the User Applications is synchronous, the socket_user_listener sends back the data returned by the called funciton. In case of socket_ADC_listener and zeroconf_listener the communication is asynchronous

5.3.2 User Application Interface

The following methods are used to receive information about the availability of the devices and the data.

class applications.pyqt_app.GUI.**GUI_Class**(*ui*, *zmq_rpc*, *GUI_name*)

register_ADC(*unique_ADC_name*, *number_of_channels*)
Registers a new ADC.

Parameters

- **unique_ADC_name** – name of the Device Application (ADC)
- **number_of_channels** – number of channels in the ADC

unregister_ADC(*unique_ADC_name*)
Unregisters an ADC.

Parameters **`unique_ADC_name`** – name of the Device Application (ADC)

`set_ADC_available`(*unique_ADC_name*)
Makes the ADC available when other Application stops using this ADC.

Parameters **`unique_ADC_name`** – name of the Device Application (ADC)

`set_ADC_unavailable`(*unique_ADC_name*)
Makes the ADC unavailable when other Application uses this ADC.

Parameters **`unique_ADC_name`** – name of the Device Application (ADC)

`update_data`(*data*, *pre_post_samples*, *offsets*)
It is used to send the acquisition data from the Server to the User Application and, depeneding on requirements of the Application, process or display the data. In case of the GUI, the data is displayed.

Parameters

- **`data`** – dictionary with GUI channels indexes as keys, containing acquisition data
- **`pre_post_samples`** – number of acquired presamples and postsamples
- **`offsets`** – difference betwenn the timestamps of the channels, with respect to the first channel – this information is used to realign the data if for any reason the value of the timestamp is not the same

5.3.3 Device Application Interface

class `nodes.adc_lib_node.server_expose.`**`ServerExpose`**(*port*, *port_server*, *pci_addr*, *trtl*, *unique_ADC_name*)

`__getattr__`(*function_name*)
If the required function is not defined here, look for it in the devices_access object

`set_server_address`(*addr*)
If address of the Server is provided by the user, this function is called from main. If address of the Server is discovered by zeroconf, this function is called by RPC call from the Server. It creates the Publisher object used for sending the notifications and acquisition data to the Server.

Parameters **`addr`** – address of the Server

class `nodes.adc_lib_node.devices_access.`**`DevicesAccess`**(*pci_addr*, *trtl*, *unique_ADC_name*)

`set_user_app_name`(*user_app_name*)
Sets the name of the User Application – this name is used to distinguish WRTD rules.

Parameters **`user_app_name`** – Name of the User Application

`set_WRTD_master`(*WRTD_master*, *trigger_type=None*, *ADC_trigger_idx=None*)
Defines if particular device works as master or slave. The master distributes the triggers, the slave receives the triggers.

Master mode:

- enables the selected trigger
- adds WRTD rule to distribute timestamps
- modifies the number of presmaples and postsamples to align data from various ADCs

Slave mode:

- disables all triggers

- adds WRTD rule to receive timestamps
- modifies the number of presmaples and postsamples to align data from various ADCs

Parameters

- **`WRTD_master`** – if True device works as master, if False device works as slave
- **`trigger_type`** – in master mode defines the type of the trigger to enable
- **`ADC_trigger_idx`** – in master mode defines the index of the trigger to enable

`configure_adc_parameter`(*function_name*, **args*)
Generic function to modify the ADC parameters.

Parameters

- **`function_name`** – the name of the function, which modifies particular ADC parameter
- **`args`** – arguments passed to the function – the type of arguments depends on the selected function

`get_current_adc_conf`()
Retrieves the configuration of the ADC.

Returns Dictionary with ADC configurations

`configure_acquisition`(*channels*)
Configures single acquisition.

Parameters **`channels`** – list of channels indexes from which the data should be retrieved

`run_acquisition`(*run*, *channels=None*)
Starts or stops continuous acquisition.

Parameters

- **`run`** – if True start acquisition, if False stop acquisition
- **`channels`** – list of channels indexes from which the data should be retrieved

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index