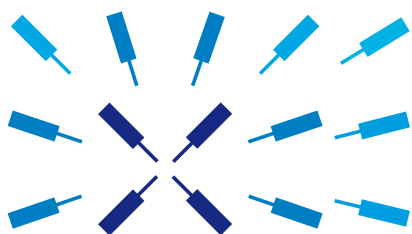


LabOne Programming Manual



Zurich
Instruments

LabOne Programming Manual

Zurich Instruments AG

Publication date Revision 58608

Copyright © 2008-2018 Zurich Instruments AG

The contents of this document are provided by Zurich Instruments AG (ZI), “as is”. ZI makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice.

LabVIEW is a registered trademark of National Instruments Inc. MATLAB is a registered trademark of The MathWorks, Inc. All other trademarks are the property of their respective owners.

Revision History

Revision 58608, 15-Dec-2018:

Update of the LabOne Programming Manual for LabOne Release 18.12.

- Updated the [Python Programming](#) chapter to reflect improvements in the [zhinst](#) LabOne Python API package and its distribution via the [PyPi](#) package repository.
- Updated the required Matlab version for the LabOne Matlab API on Linux.

Revision 53400, 17-Jun-2018:

Update of the LabOne Programming Manual for LabOne Release 18.05.

- Added a new chapter [Instrument Communication](#) which explains the various methods to obtain data from instruments.
- Major improvements to the [AWG Module](#), [PID Advisor Module](#), [Scope Module](#) sections.
- Improvements to the [Data Acquisition Module](#) including a comprehensive table of the possible signal paths that the module may be subscribed to ([Table 3.9](#)).

Revision 49900, 22-Dec-2017:

Update of the LabOne Programming Manual for LabOne Release 17.12.

- Added a new section for the new [Data Acquisition Module](#) which supersedes both the Software Trigger (Recorder) and Spectrum (ZoomFFT) Modules.
- Added a [sub-section](#) describing how to use the SW Trigger's findLevel functionality.

Revision 45917, 06-Jul-2017:

Update of the LabOne Programming Manual for LabOne Release 17.06.

- Added description of [API Level 6](#).
- Documented new API functions `setString()` and `getString()`.
- Documented new `streamingonly` and `subscribedonly` flags for `listNodes()`.
- Documented new `settingsonly` flag for `get()`.
- Corrections to parameter documentation, in particular for the [Multi-Device Synchronisation Module](#).

Revision 42453, 11-Jan-2017:

Update of the LabOne Programming Manual for LabOne Release 16.12.

- Added a new chapter for the [LabOne .NET API](#).
 - Re-worked the instrument communication section which describes how to stream data from and configure instruments. Added documentation for Data Server nodes types and a table of the available
-

instrument streaming nodes. Elaborated the description of working with the subscribe and poll commands.

- Improved the [Initializing a Connection to a Data Server](#) section for the case of using an MFLI connected via USB.
- Added new sections to the [ziCore Chapter](#) for the new Core Modules added in 16.12:
 - [PID Advisor Module](#),
 - [Scope Module](#),
 - [Impedance Module](#),
 - [Multi-Device Synchronisation Module](#),
 - [AWG Module](#).
- Removed the PLL Advisor Core Module section as the PLL Advisor Module is deprecated as of 16.12; users should use the [PID Advisor Module](#) instead.
- Updated Matlab and Python reference documentation to include Discovery documentation.
- Added a short section [Using the SW Trigger with a Digital Trigger](#) which explains the `trigger/bits` and `trigger/mask` parameters.

Revision 38200, 12-Jul-2016:

Update of the LabOne Programming Manual for LabOne Release 16.04.

- Update of Core Module parameter tables.

Revision 34390, 23-Dec-2015:

Update of the LabOne Programming Manual for LabOne Release 15.11.

- [Sweeper Module](#) parameter and description update; explanation of `sweep/inaccuracy`.
- LabOne Matlab, Python and C APIs: Documentation added describing the [APIs logging capabilities](#).
- LabOne C API Documentation added for [Core Module functionality](#).
- LabOne C API: Updates to the [Error Handling section](#) following the introduction of the `ziGetLastError` function.

Revision 31421, 8-Jul-2015:

Update of the LabOne Programming Manual for LabOne Release 15.05.

- The LabOne LabVIEW and C (ziAPI) APIs are now `ziCore`-based.
- Additions and modifications for MFLI support.

Revision 28870, 18-Mar-2015:

Update of the LabOne Programming Manual for LabOne Release 15.01.

- Added description of API Level 5.

Revision 26206, 01-Oct-2014:

Update of the LabOne Programming Manual for LabOne Release 14.08.

- Added PLL Advisor Module section to `ziCore` Modules.
- Consistency update of `ziCore` Module parameters.
- Improvements to plots in Software Trigger `ziCore` Module section.

Revision 23212, 23-Apr-2014:

- First release of the LabOne Programming Manual.
-

Table of Contents

1. Introduction	5
1.1. LabOne Programming Quick Start Guide	6
1.2. LabOne Software Architecture	8
1.3. Comparison of the LabOne APIs	10
1.4. Initializing a Connection to a Data Server	11
1.5. Compatibility	16
2. Instrument Communication	17
2.1. The Data Server's Node Tree	18
2.2. Data Streaming	22
2.3. Comparison of Data Acquisition Methods	26
2.4. Demodulator Sample Data Structure	27
2.5. Instrument-Specific Considerations	28
3. LabOne API Programming	29
3.1. An Introduction to LabOne Modules	30
3.2. Low-level LabOne API Commands	32
3.3. Common Core Module Parameters	37
3.4. AWG Module	38
3.5. Data Acquisition Module	44
3.6. Device Settings Module	59
3.7. Impedance Module	60
3.8. Multi-Device Synchronisation Module	62
3.9. PID Advisor Module	63
3.10. Scope Module	71
3.11. Sweeper Module	79
3.12. Software Trigger (Recorder) Module	90
3.13. Spectrum Analyzer Module	99
4. Matlab Programming	102
4.1. Installing the LabOne Matlab API	103
4.2. Getting Started with the LabOne Matlab API	106
4.3. LabOne Matlab API Tips and Tricks	112
4.4. Troubleshooting the LabOne Matlab API	114
4.5. LabOne Matlab API (ziDAQ) Command Reference	116
5. Python Programming	140
5.1. Installing the LabOne Python API	141
5.2. Getting Started with the LabOne Python API	144
5.3. LabOne Python API Tips and Tricks	150
5.4. LabOne Python API (ziPython) Command Reference	151
6. LabVIEW Programming	213
6.1. Installing the LabOne LabVIEW API	214
6.2. Getting Started with the LabOne LabVIEW API	216
6.3. LabVIEW Programming Tips and Tricks	221
7. .NET Programming	222
7.1. Installing the LabOne .NET API	223
7.2. Getting Started with the LabOne .NET API	224
7.3. LabOne .NET API Examples	228
8. C Programming	253
8.1. Getting Started	254
8.2. Module Documentation	256
8.3. Data Structure Documentation	386
8.4. File Documentation	445
Glossary	636
Index	642

Chapter 1. Introduction

This chapter briefly describes the different possibilities to interface with a Zurich Instruments device, other than via the LabOne User Interface. Zurich Instruments devices are designed with the concept that "the computer is the cockpit"; there are no controls on the front panel of the instrument, instead the user can configure their instrument from and stream data directly to their computer. The aim of this approach is to give the user the freedom to choose where they connect to, and how they control, their instrument.

Refer to:

- [Section 1.1](#) for a [LabOne Programming Quick Start Guide](#).
- [Section 1.2](#) for an overview of the [LabOne Software Architecture](#).
- [Section 1.3](#) for a [Comparison of the LabOne APIs](#).
- [Section 1.4](#) for help [Initializing a Connection to a Data Server](#).

Instrument Specifics

The LabOne Programming Manual is intended to be used in parallel to the corresponding user manual for the instrument you are using. Please refer to the instrument-specific manual for comprehensive documentation of its functionality and settings; a full list of settings can be found in the "Device Node Tree" Chapter.

HF2 Real-time Option

The Real-time Option (RTK) for the HF2 Series is not a PC-based interface for controlling an instrument and is documented in the HF2 User Manual.

1.1. LabOne Programming Quick Start Guide

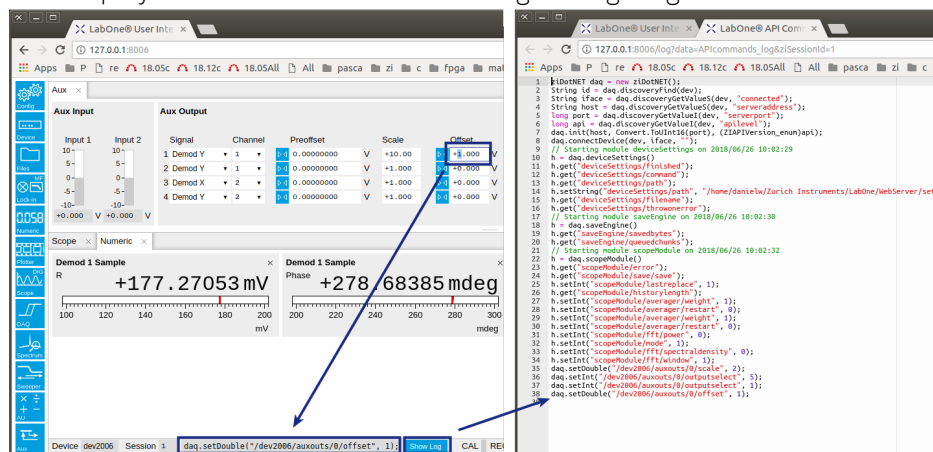
This section contains a collection of tips to help you get started programming with your instrument as quickly as possible.

Use the LabOne User Interface to develop measurement methods

LabOne's high-level measurement tools, such as the [Sweeper](#) are available in the LabOne User Interface (UI) and APIs. Since they use the same internal library, these tools have consistent behaviour across all interfaces; their parameters may first be tuned in the UI before transferring them to the API. Documentation for all the available high-level tools are provided in [Chapter 3](#).

Let the LabOne User Interface write code for you

Use the command logging functionality of the the UI to copy code from the UI's log to your Matlab, Python or .NET program. When you change a setting in the UI the corresponding API command is displayed in the status bar. If "Show Log" is clicked, then the full history of corresponding API commands is displayed. More information on finding settings is given in [Section 2.1.2](#).



Use Device Discovery to connect to your device

Device Discovery is included in every API and provides information on all the devices connected via USB or that are visible on your local network. For example, from Python: Import the module, create an instance of the Discovery tool and request discovery information on a specific device:

```
import zhinst.ziPython
dev = 'dev2006'
d = zhinst.ziPython.ziDiscovery()
props = d.get(d.find(dev))
```

Device Discovery returns a dictionary that contains connectivity information for the specified device; we can use this information to create an API session (connect to a Data Server) and subsequently connect the device on a physical interface (e.g. USB, ethernet) and start communicating with the device:

```
api_session = zhinst.ziPython.ziDAQServer(props['serveraddress'],
    props['serverport'], props['apilevel'])
api_session.connectDevice(dev, props['interfaces'][0])
api_session.getDouble('/{/}/demods/0/rate'.format(dev))
# Out: 1.81e3
```

Further explanation about creating an API session with the appropriate the Data Server is provided in [Section 1.4.1](#).

Use the distributed examples as a base for your program

Each LabOne API contains many examples to help you get started. Refer to the API-specific chapter on information on locating the examples: [Matlab](#), [Python](#), [LabVIEW](#), [.NET](#), [C](#).

Use the API's logging capabilities

The LabOne APIs write log files containing useful debugging and status information. Enable the API log to monitor the behaviour of your program and add your own log entries with the `writeDebugLog()` command. For example, in Python: Import the `ziPython` module, create an API session and enable logging with the `setDebugLevel()` command:

```
import zhinst.ziPython
api_session = zhinst.ziPython.ziDAQServer('localhost', 8004, 6)
api_session.setDebugLevel(0)
```

Then create an instance of the Sweeper and write to the log:

```
h = api_session.sweep()
api_session.writeDebugLog(0, 'Will now configure and start the sweeper...')
h.set('sweep/device', 'dev2006')
h.execute()
```

These commands generate the log:

```
Will log to directory '/tmp/ziPythonLog_danielw'
11:55.25.805 [0] [status] Opening session: 127.0.0.1
11:55.25.805 [1] [trace] Will now configure and start the sweeper...
11:55.30.877 [2] [debug] Sweep execute, averaging 1 samples 0.001 s or 5 x tc,
    settling 0s or 5 x tc (0.01)
11:55.30.886 [3] [debug] Oscillator index 0 for path oscs/0/freq
11:55.30.887 [4] [warning] Sweeper will run in slower synchronous set mode. Enable
    controlled demodulators to get fast mode.
11:55.30.888 [5] [debug] Sweep fast: false, bandwidth control: 2, bw: 2000
11:55.30.889 [6] [debug] Used settlingTimeFactor 9.99805
```

See the API-specific chapter for more details: [Matlab](#), [Python](#), [C](#).

Use the API utility functions

The APIs are distributed with utility functions that replicate functionality incorporated in the UI. For example, the Matlab and Python APIs have utility functions to convert a demodulator's time constant to its corresponding 3dB bandwidth (`tc2bw`).

Load LabOne User Interface settings files from the APIs

The [XML](#) files used for device settings can not only be loaded and saved from the LabOne User Interface but from any of the APIs. See [Section 3.6, Device Settings Module](#) for more information.

1.2. LabOne Software Architecture

Zurich Instruments devices use a server-based connectivity methodology. Server-based means that all communication between the user and the instrument takes place via a computer program called a server, the Data Server. The Data Server recognizes available instruments and manages all communication between the instrument and the host computer on one side, and communication to all the connected clients on the other side. This allows for:

- A multi-client configuration: Multiple interfaces (even from multiple computers on the network) can access the settings and data on an instrument. Settings are synchronized between all interfaces by the single instance of the Data Server.
- A multi-device setup: Any of the Data Server's clients can access multiple devices simultaneously.

This software architecture is organized in layers, see [Figure 1.1](#) for a schematic of the software layers.

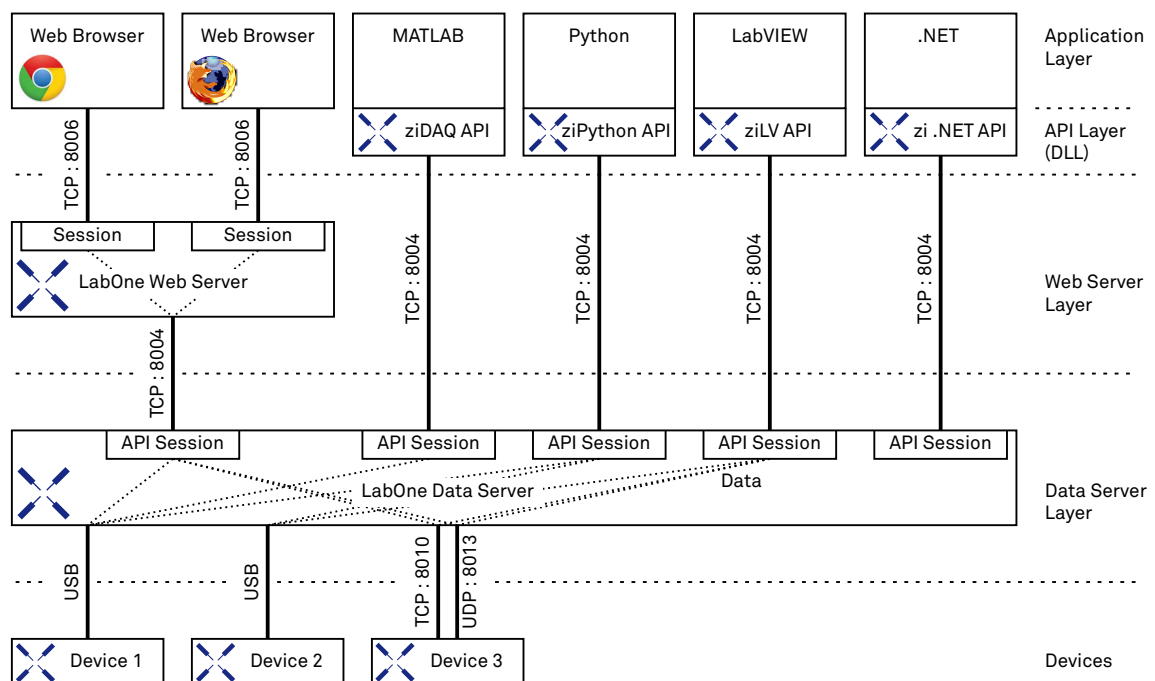


Figure 1.1. LabOne Software Architecture. The above diagram depicts the software architecture when using UHF, HDAWG and HF2 Instruments. In the case of MF Instruments the Data Server runs on the device itself instead of on a PC; only one MF device can be accessed from the Data Server. Web Server and API usage for the MF is analogous to that of other instruments.

First, we briefly explain some terminology that is used throughout this manual.

- **Host computer:** The computer where the Data Server is running and that is directly connected to the instrument. Multiple remote computers on a local area network can access the instrument by creating an API connection to the Data Server running on the host computer.
- **Data Server:** A computer program that runs on the host computer and manages settings on, and data transfer to and from instruments by receiving commands from clients. It always has the most up-to-date configuration of the device and ensures that the configuration is synchronized between different clients.
- **ziServer.exe:** The Data Server that handles communication with HF2 Instruments.

- `ziDataServer.exe`: The Data Server that handles communication with HDAWG, MF and UHF Instruments. Note, in the case of MFLI Instruments the Data Server runs on the instrument itself.
- Remote computer: A computer, available on the same network as the host computer, that can communicate with an instrument via the Data Server program running on the host.
- Client: A computer program that communicates with an instrument via the Data Server. The client can be running either on the host or the remote computer.
- API (Application Programming Interface): a collection of functions and data structures which enable communication between software components. In our case, the various APIs (e.g., LabVIEW, Matlab®) provide functions to configure instruments and receive measured experimental data.
- Interface: Either a client or an API.
- GUI (Graphical User Interface): A computer program that the user can operate via images as opposed to text-based commands.
- LabOne User Interface: The browser-based user interface that connects to the Web Server.
- LabOne Web Server: The program that generates the browser-based LabOne User Interface.
- `ziControl`: The standard GUI shipped for use with HF2 Instruments (before software release 15.11). HF2 support was added to the LabOne User Interface for devices with the WEB Option installed in LabOne software release 15.11 .
- `ziCore`: The internal core library upon which many APIs are based, see [Chapter 3](#) for more information.
- Modules: `ziCore` software components that provide a unified interface to APIs to perform a specific high-level common task such as sweeping data.

1.3. Comparison of the LabOne APIs

The various software interfaces available in LabOne allow the user to pick a programming environment they are familiar with to achieve fast results. All other things being equal, here is a brief discussion of the merits of each interface.

- The [LabVIEW interface](#) allows for quick and efficient implementation of virtual instruments that run independently. These can easily be integrated in existing experiment control performed in LabVIEW. This interface requires a National Instruments LabVIEW license and LabVIEW 2009 (or higher).
- The [Matlab® interface](#) allows the user to directly obtain measurement data within the Matlab programming environment, where they can make use of the many built-in functions available. This interface requires a MathWorks Matlab license, but no additional Matlab Toolboxes.
- The [Python interface](#) allows the user to directly obtain measurement data within python. Python is available as free and open source software; no license is required to use it.
- The [.NET interface](#) allows the user to directly obtain measurement data within the .NET programming framework using the C#, Visual Basic or #F programming languages. To use the .NET API a Microsoft Visual Studio installation is required.
- The [C API](#), [ziAPI](#), is a very versatile interface that will run on most platforms. However, since C is a low-level programming language, the development cycle is slower than with the other programming environments.
- The text-based interface (HF2 Series only) allows the user to manually connect to the HF2 Data Server in a console via telnet. While this interface is a very useful tool for HF2 programmers to verify instrument configuration set by other interfaces, it is limited in terms of performance and maximum demodulator sample rate. See the HF2 User Manual for more details.

Note

From LabOne Release 15.05 onwards the Sweeper and Software Trigger [Modules](#) are also available in the LabVIEW and C APIs and from 16.12 onwards all Modules are available. All modules were previously available in the Matlab and Python LabOne APIs.

1.4. Initializing a Connection to a Data Server

As described in [Section 1.2](#) an API client communicates with an instrument via a data server over a TCP/IP socket connection. As such, the first step towards communicating with an instrument is initializing an API session to the correct data server for the target device.

The choice of data server depends on the device class and on the network topology. HF2 instruments operate via a different data server program than HDAWG, MF and UHF instruments. Users of MF instruments should be aware that the data server runs on the MF instrument itself and not on a separate PC. Finally, in the case of MF instruments, the way to connect to the data server depends on the interface (USB or 1GbE). In all cases, the desired data server is specified by providing three parameters:

- the data server host's address (hostname),
- the data server port,
- the API level to use for the session.

1.4.1. Specifying the Data Server Hostname and Port

For users working with a single device, this section describes how to quickly connect to the correct data server by manually specifying the required data server's hostname and port and the required API Level. Each API has a connect function which takes these three parameters in order to initialize an API session, for example, in the LabOne Matlab API:

```
>>> ziDAQ('connect', serverHostname, serverPort, apiLevel);
```

Data Server Port

A LabOne API client connects to the correct Data Server for their instrument by specifying the appropriate port. By default, the data server programs for HDAWG, MF and UHF Instruments listen to port 8004 for API connections and the data server program for HF2 instruments listens to port 8005. The value of the port that the data server listens to can be changed using the `--port` command-line option when starting the data server.

Data Server Hostname (HDAWG, HF2 and UHF Instruments)

In the simplest configuration for HDAWG, HF2 and UHF instruments, the instrument is attached to the same PC where both the data server and API client are running. Since the API client is running on the same PC as the data server, the `'localhost'` (equivalently, `'127.0.0.1'`) should be specified as the data server address, [Figure 1.2](#).

The API client may also connect to a data server running on a different PC from the client. In this case, the data server address should be the IP address (or hostname, if available) of the PC where the data server is running. Note, remote data server access is not enabled by default and the data server must be configured in order to listen to non-localhost connections by either enabling the `--open-override` command-line option when starting the data server or by setting the value of the server node `/zi/config/open` to 1 on a running data server (clearly only possible from a client running on the localhost). See [Section 2.1](#) for more information on nodes.

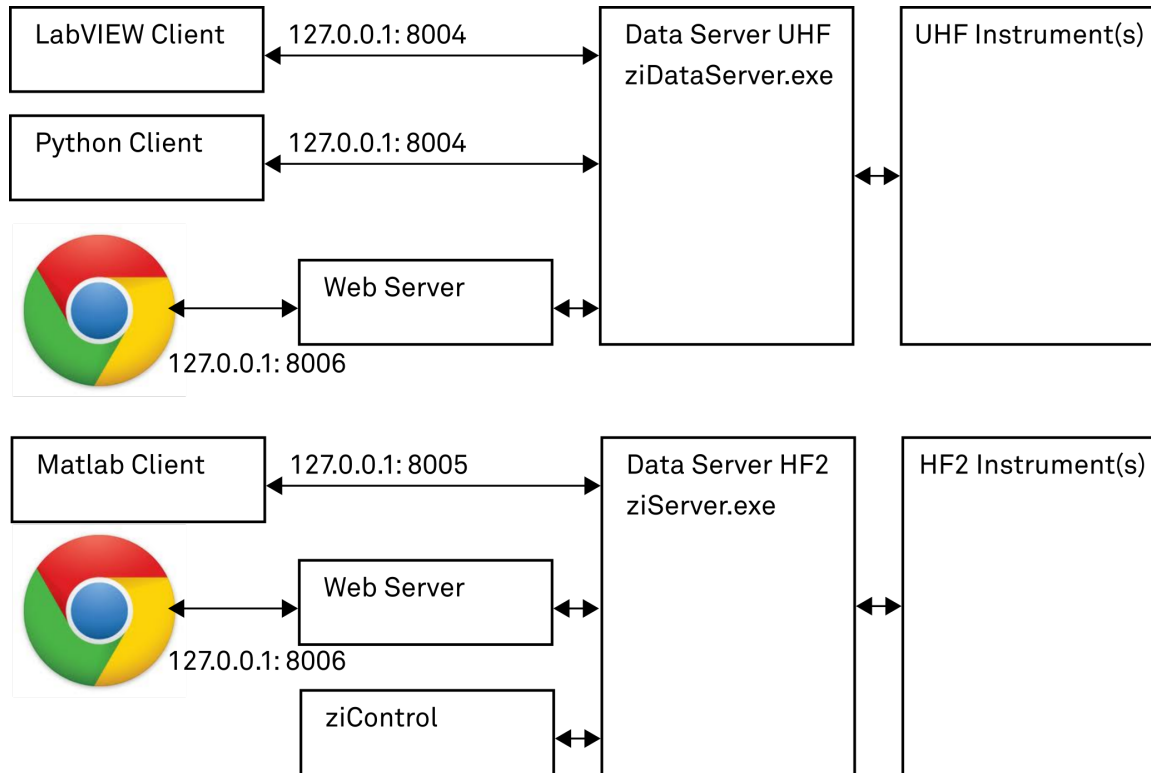


Figure 1.2. Server address and port handling for HDAWG, HF2 and UHF instruments for the case where the API client and data server are running on the same PC. In this case the server hostname is `localhost` and the default port value is 8004 for HDAWG and UHF Instruments and 8005 for HF2 Instruments.

Data Server Hostname (MF Instruments)

In the case of MF instruments the data server is running on the instrument itself and as such an API client from a PC is always accessing the data server remotely. Thus, in this case the data server hostname is the value of the instrument's hostname. This will be the same hostname (but not port) that is used to run the LabOne User Interface in a web browser (when the Web Server is running on the MF instrument), see [Figure 1.3](#).

As described in more detailed in the Getting Started chapter of the MFLI User Manual, the MF instrument hostname can either be its instrument serial of the form `mf-dev3001`, or its IP address. The former is however only valid if the MF instrument is connected to a LAN with domain name system via 1GbE. If it's connected via the USB interface, finding out the IP address is easiest by using the Start Menu Entry "LabOne User Interface MF USB" and then copying the IP address from the browser's address bar.

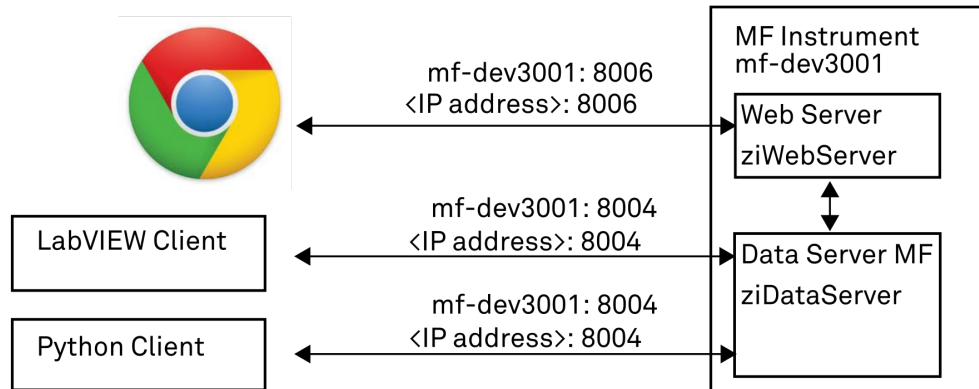


Figure 1.3. Server address and port handling on MF Instruments. The data server is running on the instrument and the server hostname is the same as the instrument's hostname. Using the instrument of the form `mf-dev3001` as a hostname is only applicable when using the 1GbE interface. The default data server port is 8004 for MF Instruments.

API Level and Connectivity Examples

The last parameter to specify, the API level, specifies the version of the API to use for the session. In short, an API Level of 1 must be used for HF2 devices and an API Level 6 is recommended for other instruments. Since the default API Level is 1, it is necessary to specify this parameter for UHF and MF instruments. A more detailed explanation of API Levels is provided in [Section 1.4.2](#).

For example, to initialize a session to the HF2's data server running on the `localhost` with the LabOne Python API, the following commands should be used:

```
>>> import zhinst.ziPython
>>> daq = zhinst.ziPython.ziDAQServer('localhost', 8005, 1)
```

and in order to connect to the data server running on the MF instrument connected via 1GbE with device serial `'dev3001'` with the LabOne Matlab API:

```
>> ziDAQ('connect', 'mf-dev3001', 8004, 6)
```

On an MF instrument connected via USB, the device serial cannot be directly used as the hostname, instead one needs to use the instrument's IP address. Unless this is known beforehand, it can be determined by the network discovery functionality of the API. The following python example shows how this can be done:

```
>>> import zhinst.ziPython
>>> d = zhinst.ziPython.ziDiscovery()
>>> d.find('mf-dev3001')
>>> devProp = d.get('mf-dev3001')
>>> daq = zhinst.ziPython.ziDAQServer(devProp['serveraddress'],
>>> devProp['serverport'], 6)
```

Working in a Multi-threaded Program

It is important to note that API session objects are not thread-safe, i.e., one instance of the session should not be used by multiple threads. If you want to use a LabOne API in a multi-threaded program, one session should be used for each thread that is communicating with the Data Server or a mutual exclusion must be implemented.

1.4.2. LabOne API Levels

All of the LabOne APIs are based on an internal core API. Needless to say, we try as hard as possible to make any improvements in our core API backwards compatible for the convenience of our users. We take care that existing programs do not need to be changed upon a new software release.

Occasionally, however, we do have to make a breaking change in our API by removing some old functionality. This old functionality is, however, phased out over several software releases. First, the functionality is marked as deprecated and the user is informed via a depreciation warning (this can be turned off). This indicator warns that this function may be unsupported in the future. If we have to break some functionality we use a so-called **API level**.

With support of new devices and features we need to break functionality on the ziAPI.h e.g. data returned by poll commands. In order to still support the old functionality we introduced API levels. If a program is only using old functionality the API level 1 (default) can be used. If a user needs new functionality, they need to use a higher API level. This will usually need some changes on the existing code.

The current available API levels are:

- API Level 1: HF2 support, basic UHF support.
- API Level 4: UHF support, timestamp support in poll, PWA, name clean-up.
- API Level 5: Introduction of scope offset for extended (non-hardware) scope inputs (UHF, MF Instruments).
- API Level 6: Timestamp support in poll for nodes that return a byte array.

Note that Levels 2 and 3 are used only internally and are not available to the general public.

Note

The HF2 Series only supports API Level 1.

Note

New HDAWG, MF and UHF API users are recommended to use API Level 6.

API Level 4 Features

The new features in API Level 4 are:

- Timestamps are available for any settings or data node (that is either integer or float).
- Greatly improved Scope data transfer rates (and new Scope data structure).
- Greatly improved UHF Boxcar and PWA support.

API Level 5 Features

API Level 5 was introduced in LabOne Release 15.01 to accommodate a necessary change in the Scope data structure:

- The Scope data structure was extended with the new field "channeloffset" which contains the offset value that must be added to the scaled wave value in order to obtain the physical value recorded by the scope. For previous hardware scope "inputselects" there is essentially no change, since their offset is always zero. However, for the extended values of "inputselects", such as PID Out value, (available with the DIG option) the offset is determined by the values of "limitlower" and "limitupper" configured by the user.

API Level 6 Features

API Level 6 was introduced in LabOne Release 17.06 to make the behavior of poll for nodes that return a byte array consistent with nodes that return integer and float data:

- Timestamps are returned for all byte array nodes.
- New commands `setString` and `getString` are available and should be used instead of `setByte` and `getByte`.

1.5. Compatibility

Controlling an instrument requires the combination of several software components: The instrument's firmware, a Data Server and an API. In general, whenever possible, it is recommended to use the latest (and same) software release version (e.g., "17.06") of all these components. If you are bound to a certain version for technical reasons, then it is recommended to use the same version of all components. However, this is not strictly necessary in all cases. If it is absolutely necessary to mix versions, this section explains how to verify whether different versions of various software components may be mixed with each other.

1.5.1. API and Data Server Compatibility

Although it is recommended to use the same software release version (e.g. "17.06") of both API and Data Server, it is not strictly necessary. The interface between API and Data Server remains the same between versions. However, there may be a change in some [nodes](#) that effects specific functionality.

If you do need to mix versions, then please check the Release Notes (included in a LabOne installation) to see if the functionality you require has changed. If so, then the same version of API and Data Server must be used. Otherwise, it is possible to mix versions. If after checking the Release Notes you are still not sure, then please contact Zurich Instruments customer support.

All the LabOne APIs have a utility function to check whether the API being used is the same version as the Data Server it is connected to, e.g., `api_server_version_check()` in the Python API and `ziApiServerVersionCheck()` in the Matlab API.

Chapter 2. Instrument Communication

This section describes the main concepts in LabOne software that allow high-speed data acquisition and guides the user to the best acquisition method for their measurement task. It is divided into sub-sections as following:

- [Section 2.1](#) explains how device settings and data are organized and accessible in [The Data Server's Node Tree](#).
- [Section 2.2](#) explains Zurich Instruments' [Data Streaming](#) concept for data acquisition.
- [Section 2.3](#) compares the methods available for [Comparison of Data Acquisition Methods](#).
- [Section 2.5](#) documents some [Instrument-Specific Considerations](#).

2.1. The Data Server's Node Tree

This section provides an overview of how an instrument's configuration and output is organised by the Data Server.

All communication with an instrument occurs via the Data Server program the instrument is connected to (see [Section 1.2](#) for an overview of LabOne's software components). Although the instrument's settings are stored locally on the device, it is the Data Server's task to ensure it maintains the values of the current settings and makes these settings (and any subscribed data) available to all its current clients. A client may be the LabOne User Interface or a user's own program implemented using one of the LabOne Application Programming Interfaces, e.g., Python.

The instrument's settings and data are organized by the Data Server in a file-system-like hierarchical structure called the node tree. When an instrument is connected to a Data Server, its device ID becomes a top-level branch in the Data Server's node tree. The features of the instrument are organised as branches underneath the top-level device branch and the individual instrument settings are leaves of these branches.

For example, the auxiliary outputs of the instrument with device ID "dev2006" are located in the tree in the branch:

```
/DEV2006/AUXOUTS/
```

In turn, each individual auxiliary output channel has its own branch underneath the "AUXOUTS" branch.

```
/DEV2006/AUXOUTS/0/  
/DEV2006/AUXOUTS/1/  
/DEV2006/AUXOUTS/2/  
/DEV2006/AUXOUTS/3/
```

Whilst the auxiliary outputs and other channels are labelled on the instrument's panels and the User Interface using 1-based indexing, the Data Server's node tree uses 0-based indexing. Individual settings (and data) of an auxiliary output are available as leaves underneath the corresponding channel's branch:

```
/DEV2006/AUXOUTS/0/DEMOMODESELECT  
/DEV2006/AUXOUTS/0/LIMITLOWER  
/DEV2006/AUXOUTS/0/LIMITUPPER  
/DEV2006/AUXOUTS/0/OFFSET  
/DEV2006/AUXOUTS/0/OUTPUTSELECT  
/DEV2006/AUXOUTS/0/PREOFFSET  
/DEV2006/AUXOUTS/0/SCALE  
/DEV2006/AUXOUTS/0/VALUE
```

These are all individual node paths in the node tree; the lowest-level nodes which represent a single instrument setting or data stream. Whether the node is an instrument setting or data-stream and which type of data it contains or provides is well-defined and documented on a per-node basis in the Reference Node Documentation section in the relevant instrument-specific user manual. The different properties and types are explained in [Section 2.1.1](#).

For instrument settings, a Data Server client modifies the node's value by specifying the appropriate path and a value to the Data Server as a (path, value) pair. When an instrument's setting is changed in the LabOne User Interface the path and the value of the node that was changed are displayed in the Status Bar in the bottom of the Window. This is described in more detail in [Section 2.1.2](#).

Module Parameters

LabOne Core Modules, such as the Sweeper, also use a similar tree-like structure to organize their parameters. Please note, however, that module nodes are not visible in the Data Server's node tree; they are local to the instance of the module created in a LabOne client and are not synchronized between clients.

2.1.1. Node Properties and Data Types

A node may have one or more of the following properties:

Read	Data can be read from the node.
Write	Data can be written to the node.
Setting	A node with write attribute corresponding to an instrument configuration. The data in these nodes will be saved to and loaded from LabOne XML settings files.
Streaming	A node with the read attribute that provides instrument data, typically at a user-configured rate. The data is usually a more complex data type, for example demodulator data is returned as ZIDemodSample . A full list of streaming nodes is available in Table 2.1 .

A node may contain data of the following types:

Integer	Integer data.
Double	Double precision floating point data. Note, that the actual value on the device may only be calculated in single precision.
String	A string array.
Integer (enumerated)	As for Integer, but the node only allows certain values.
Complex data type	For example, ZIDemodSample . These custom data types are structures whose fields contain the instrument output, a timestamp and other relevant instrument settings such as the demodulator oscillator frequency. Documentation of custom data types is available in the C Programming chapter .

2.1.2. Exploring the Node Tree

In the LabOne User Interface

A convenient method to learn which node is responsible for a specific instrument setting is to check the Command Log history in the bottom of the LabOne User Interface. The command in the Status Bar gets updated every time a configuration change is made. [Figure 2.1](#) shows how the equivalent Matlab command is displayed after modifying the value of the auxiliary output 1's offset. The format of the LabOne UI's command history can be configured in the Config Tab (Matlab, Python and .NET are available). The entire history generated in the current UI session can be viewed by clicking the "Show Log" button.

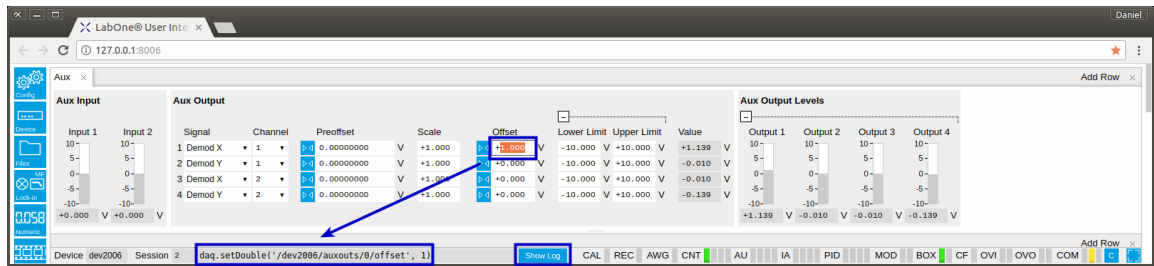


Figure 2.1. When a device's configuration is modified in the LabOne User Interface, the Status Bar displays the equivalent command to perform the same configuration via a LabOne programming interface. Here, the Matlab code to modify auxiliary output 1's offset value is provided. When "Show Log" is clicked the entire configuration history is displayed in a new browser tab.

In the Instrument-specific User Manual

Each instrument user manual has a "Device Node Tree" chapter that contains complete reference documentation of every node available on the device. This documentation may be explored by branch to obtain a complete overview of which settings are available on the instrument.

In a LabOne Programming Interface

A list of nodes (under a specific branch) can be requested from the Data Server in an API client using the `listNodes` command (Matlab, Python) or `ziAPIListNodes()` function (C API). Please see each API's command reference for more help using the `listNodes` command. To obtain a list of all the nodes that provide data from an instrument at a high rate, so-called streaming nodes, the `streamingonly` flag can be provided to `listNodes`. More information on data streaming and streaming nodes is available in [Section 2.2](#).

The detailed descriptions of nodes that is provided in the instrument-specific user manual section "Reference Node Documentation" is accessible directly in the LabOne Matlab or Python programming interfaces using the "help" command. The `help` command is `daq.help(path)` in Python and `ziDAQ('help', path)` in Matlab. The command returns a description of the instrument node including access properties, data type, units and available options. The "help" command also handles wildcards to return a detailed description of all nodes matching the path. An example is provide below.

```
daq = zhinst.ziPython.ziDAQServer('localhost', 8004, 6)
daq.help('/dev2006/auxouts/0/offset')
# Out:
# /DEV2006/AUXOUTS/0/OFFSET
# Add the specified offset voltage to the signal after scaling. Auxiliary Output
# Value = (Signal+Preoffset)*Scale + Offset
# Properties: Read, Write, Setting
# Type: Double
# Unit: V
```

2.1.3. Data Server Nodes

The Data Server has nodes in the node tree available under the top-level `/ZI/` branch. These nodes give information about the version and state of the Data Server the client is connected to. For example, the nodes:

- `/ZI/VERSION`

- /ZI/REVISION

are read-only nodes that contain information about the release version and revision of the Data Server. The nodes under the /ZI/DEVICES/ list which devices are connected, discoverable and visible to the Data Server.

The nodes:

- /ZI/CONFIG/OPEN
- /ZI/CONFIG/PORT

are settings nodes that can be used to configure which port the Data Server listens to for incoming client connections and whether it may accept connections from clients on hosts other than the localhost. See [Section 1.4](#) for more information about specifying the Data Server host and port.

Nodes that are of particular use to programmers are:

- /ZI/DEBUG/LOGPATH - the location of the Data Server's log in the PC's filesystem,
- /ZI/DEBUG/LEVEL - the current log-level of the Data Server (configurable; has the Write attribute),
- /ZI/DEBUG/LOG - the last Data Server log entries as a string array.

For documentation of all Data Server nodes see the /ZI/ section in the Reference Node Documentation section in the instrument-specific user manual.

2.2. Data Streaming

Zurich Instrument's Data Servers and devices allow high-speed data acquisition using the "data streaming" concept. The term "data streaming" refers to the fact that the discrete values of a device's output are continuously pushed at a high rate from the device to an API client (via the device's physical connection and Data Server) analogously to media streaming where, for example, video is continuously streamed from one computer to another over the internet.

Data streaming is only available for device outputs such as demodulator and pulse counters where it is relevant to acquire the instrument's discrete data at a very high time resolution. Settings nodes, for example, are not streamed but rather sent upon request. Some device outputs additionally allow their data stream to be gated based on the value of another device signal, such as a trigger input, for example. This allows even higher data transfer rates for short bursts than when data is continuously sent from the device.

To optimize the bandwidth of the instrument's physical connection to the Data Server (e.g., USB), streaming data for all nodes is not always sent by the device, rather each node must be enabled at the desired rate by a client. When streaming data from a device output is enabled then it always sent from the device to the Data Server. It is not sent from the Data Server to the client, however, until the API client explicitly requests the data by "subscribing" to the data server node providing the streaming data.

The advantage of Zurich Instruments' streaming concept is that it allows extremely high data acquisition rates. Whereas "fixed rate" data transfer or "fixed buffer size" data transfer (for all nodes) allow very simple interfaces for data acquisition, they do not optimise the available interface bandwidth since low update rates must be imposed to ensure that data loss does not occur in all situations. Users who prefer a fixed rate or fixed size buffer transfer may use the [Data Acquisition Module](#) which adds an additional layer in software on top of data streaming that emulates these kinds of transfer.

In general, unless extremely fast update rates or high performance data transfer is required by the client, the [Data Acquisition Module](#) is the recommended method to obtain data (as apposed to the low-level [subscribe and poll commands](#)). This is due to the additional complexities involved when working directly with "raw" streaming data since users must:

- Be aware that [data loss](#) may occur and must be correctly handled in their API client program.
- Organize data that may be split across subsequent poll commands.
- [Align/interpolate](#) streaming data from multiple nodes onto a common time grid (if required).

The following sections explain these points in further detail, users who use the data for which [Data Acquisition Module](#) may prefer to skip ahead [Section 2.3](#).

2.2.1. Streaming Nodes

When streaming nodes are recorded directly via the the low-level [subscribe and poll](#) commands they continuously deliver either:

- Continuous equidistantly spaced data in time, e.g., DEMODS/0/SAMPLE or CNTS/0/SAMPLE,
- Continuous non-equidistantly spaced data in time, e.g., BOXCARS/0/SAMPLE when the boxcar is configured with a external reference-controlled oscillator (but this is somewhat of a special case).
- Non-continuous framed "block" data that consist of chunks of data, e.g., SCOPES/0/WAVE.

The qualifier "continuous" requires more explanation. Each sample point is a discrete data point sent from the device. By continuous it is meant that the values are not only continually sent, but

also that all the data for that device unit/output is sent without gaps in time. Block data is on the other hand not continuous; it is a single burst of data from a device unit (e.g. scope) that provides data at a very high data rate.

Nodes that deliver high-speed streaming data have the streaming property set (see [Section 2.1.1](#)). This property can be used with the `listNodes` command in order to list all the streaming nodes available on a particular device. For example, in Python:

```
daq = zhinst.ziPython.ziDAQServer('localhost', 8004, 6)
from zhinst.ziPython import ziListEnum
flags = ziListEnum.recursive | ziListEnum.absolute | ziListEnum.streamingonly
print(int(ziListEnum.streamingonly), flags)
# Out:
# 16, 19
daq.connectDevice('dev2006', 'usb')
daq.listNodes('/dev2006/*/0', flags)
# Out:
# ['/DEV2006/AUCARTS/0/SAMPLE',
#  '/DEV2006/AUPOLARS/0/SAMPLE',
#  '/DEV2006/AUXINS/0/SAMPLE',
#  '/DEV2006/BOXCARS/0/SAMPLE',
#  '/DEV2006/CNTS/0/SAMPLE',
#  '/DEV2006/DEMODS/0/SAMPLE',
#  '/DEV2006/DIOS/0/INPUT',
#  '/DEV2006/INPUTPWAS/0/WAVE',
#  '/DEV2006/OUTPUTPWAS/0/WAVE',
#  '/DEV2006/PIDS/0/STREAM/ERROR',
#  '/DEV2006/PIDS/0/STREAM/SHIFT',
#  '/DEV2006/PIDS/0/STREAM/VALUE',
#  '/DEV2006/SCOPES/0/STREAM/SAMPLE',
#  '/DEV2006/SCOPES/0/WAVE']
```

The table below gives an overview of the available streaming nodes on different devices.

Table 2.1. Device streaming nodes. Their availability depends on the device class (e.g. MF) and the option set installed on the device.

Device Node Path	Availability	Type	Description
AUCARTS/n/SAMPLE	UHF	Continuous	The output samples of a Cartesian Arithmetic Unit
AUPOLARS/n/SAMPLE	UHF	Continuous	The output samples of a Polar Arithmetic Unit
AUXINS/n/SAMPLE	All instruments	Continuous	The auxiliary input samples. Typically not used; these values are included as fields in demodulator samples where available.
BOXCARS/n/SAMPLE	UHF with Box Option	Continuous	The output samples of a boxcar.
CNTS/n/SAMPLE	UHF or HDAWG with CNT Option	Continuous	The output samples of a counter unit.
DEMODS/n/SAMPLE	UHFLI, HF2LI, MFLI	Continuous	The output samples of a demodulator.
DIOS/n/INPUT	All instruments	Continuous	The DIO connector input values. Rarely used; the input values are included as a field in demodulator samples where available.

Device Node Path	Availability	Type	Description
IMPS/n/SAMPLE	MFIA, MFLI with IA Option	Continuous	The output samples of an impedance channel.
INPUTPWAS/n/WAVE	UHF with BOX Option	Block	The value of the input PWA.
OUTPUTPWAS/n/WAVE	UHF with BOX Option	Block	The value of the output PWA.
PIDS/n/STREAM/ERROR	UHF or MF with PID Option	Continuous	The error value of a PID.
PIDS/n/STREAM/SHIFT	UHF or MF with PID Option	Continuous	The shift of a PID; the difference between the center and the the output value.
PIDS/n/STREAM/VALUE	UHF or MF with PID Option	Continuous	The output value of the PID.
SCOPES/n/STREAM/SAMPLE	UHF or MF with DIG Option	Block	Scope values as a continuous block streaming node.
SCOPES/n/WAVE	All instruments	Block	Scope values as a triggered block streaming node.
TRIGGERS/STREAMS/n/SAMPLE	HDAWG	Block	Trigger values.

2.2.2. Alignment of Streaming Node Data

In general, streaming nodes deliver equidistantly-spaced samples in time (assuming no sample loss has occurred). However, different streaming nodes have different timestamp grids, this is best explained in [Figure 2.2](#).

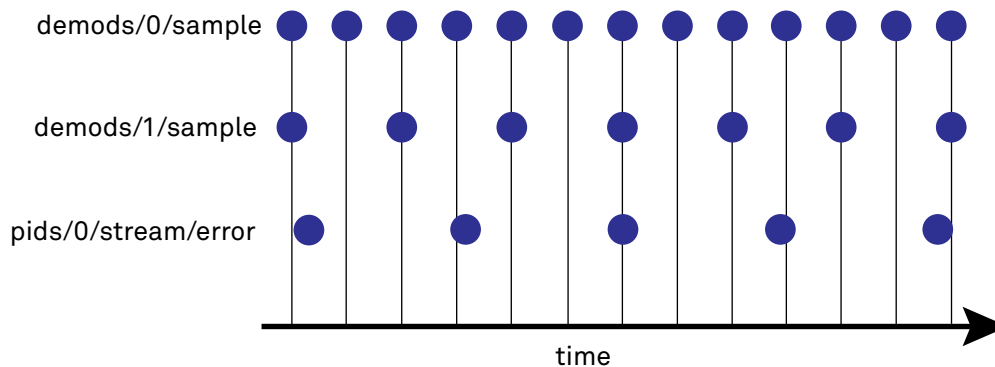


Figure 2.2. Samples from different streaming nodes configured with different rates: Demod 1 at $2N$ kSamples/s, Demod 2 at N kSamples/s and PID Error 1 at M kSample/s (N not divisible by M). Although each stream consists of equidistantly samples in time, the sample timestamps from different streams are not necessarily aligned due to the different sampling rates.

2.2.3. Data Loss

While streaming nodes deliver equidistantly sampled data in time (with the exception of a boxcar output based on an external reference controlled oscillator), they may be subject to data loss (also

called "sampleloss"). This refers to the loss of data between instrument and Data Server over the physical interface. Since data streaming is optimized for transfer speed, there is no resend mechanism available that would automatically request that missing data be transferred again to the Data Server; it is lost and this case must be handled appropriately in API programs. In the case of data loss, the returned data will simply be missing, i.e., data will no longer be equidistantly sampled in time due to the missing samples. As such, to check for data loss on streaming node data the user is advised to calculate the difference between timestamps and verify that all differences correspond to the expected difference as defined by the configured data streaming rate.

Data rates are not limited or throttled by the instrument to ensure that data loss does not occur; the user is responsible to configure streaming data rates for the required nodes such that data loss does not occur on their system. Data rates are not artificially limited due to the fact that continuous and hardware triggered data acquisition may be combined. If the cumulative data sent by the instrument over the physical interface exceeds the available bandwidth then data loss will occur. The maximum available bandwidth of the physical interface is influenced by the following factors:

- Choice of physical interface (USB, 1GbE).
- Additional load on the interface (Ethernet / USB hub) from other network traffic or devices.
- Speed of the PC where the Data Server is running.
- For the case of 1GbE interface only: The PC's network card.
- MF Instruments Only: Whether the Data and/or Web Servers are running on the instrument or on the PC.

2.3. Comparison of Data Acquisition Methods

In this section we briefly compare the methods available to obtain data from continuous streaming instrument nodes (not for block streaming nodes, see [Scope Data](#)). Which method is most appropriate depends on the requirements of the specific application. [Table 2.2](#) provides a top-level overview.

Table 2.2. Comparison of data acquisition methods available in the LabOne APIs.

Method	Good for:	Not appropriate for:
The getSample function	1. Simple single-shot measurements of demodulator data.	1. Non-demodulator streaming nodes. 2. Continuous data acquisition. 3. Triggered data acquisition.
Data Acquisition Module in Triggered Mode	1. Triggered data acquisition. 2. Aligned data from multiple streams.	
Data Acquisition Module in Continuous Mode	1. Continuous data acquisition. 2. Aligned data from multiple streams.	
The poll and subscribe functions	1. Extremely high performance.	1. Data from between different streaming nodes may not be aligned by timestamp. 2. Data alignment between polls not guaranteed.

Scope Data

It is recommended to use the [Scope Module](#) for acquiring scope data (from the node SCOPES/n/WAVE). Although scope data can be acquired using the low-level [subscribe](#) and [poll](#) commands, the [Scope Module](#) provides additional functionality such as multiple block assembly (SCOPES/n/WAVE is a "block" streaming node, cf [Section 2.1.1](#)) and FFT of the data.

2.4. Demodulator Sample Data Structure

An instrument's demodulator data is returned as a data structure (typically a `struct`) with the following fields (regardless of which API Level is used):

<code>timestamp</code>	The instrument's timestamp of the measured demodulator data <code>uint64</code> . Divide by the instrument's clockbase (<code>/dev123/clockbase</code>) to obtain the time in seconds.
<code>x</code>	The demodulator <code>x</code> value in Volts [<code>double</code>].
<code>y</code>	The demodulator <code>y</code> value in Volts [<code>double</code>].
<code>frequency</code>	The current frequency used by the demodulator in Hertz [<code>double</code>].
<code>phase</code>	The oscillator's phase in Radians (not the demodulator phase) [<code>double</code>].
<code>auxin0</code>	The auxiliary input channel 0 value in Volts [<code>double</code>].
<code>auxin1</code>	The auxiliary input channel 1 value in Volts [<code>double</code>].
<code>bits</code>	The value of the digital input/output (DIO) connector. [<code>integer</code>].
<code>time.dataloss</code>	Indicator of sample loss (including block loss) [<code>bool</code>].
<code>time.blockloss</code>	Indication of data block loss over the socket connection. This may be the result of a too long break between subsequent poll commands [<code>bool</code>].
<code>time.invalidtimestamp</code>	Indication of invalid time stamp data as a result of a sampling rate change during the measurement [<code>bool</code>].

Note

[Chapter 8](#) contains details of other data structures.

2.5. Instrument-Specific Considerations

This section describes some instrument-specific considerations when programming with the LabOne APIs.

2.5.1. MF-Specific Considerations

Identifying which Data Server the MF device is connected to

If `/DEV..../SYSTEM/ACTIVEINTERFACE` has the value "pcie" the device is connected via the Data Server running locally on the MF instrument itself. If it has the value "1GbE" it is connected via a Data Server running on a PC.

2.5.2. UHF-Specific Considerations

UHF Lock-in Amplifiers perform an automatic calibration 10 minutes after power-up of the Instrument. This internal calibration is necessary to achieve the specifications of the system. However, if necessary, it can be ran manually by setting the device node `/DEV..../SYSTEM/CALIB/CALIBRATE` to 1 and then disabled using the `/DEV..../SYSTEM/CALIB/AUTO` node.

The calibration routine takes about 200 ms and during that time the transfer of measurement data will be stopped on the Data Server level. If a [ziAPI](#) (LabOne C API) or [LabVIEW](#) client is polling data during this time, the user will experience data loss; ziAPI has no functionality to deal with such a streaming interrupt. Clients polling data will be informed of data loss, which allows the user to ignore this data.

Please see the UHF User Manual for more information about device calibration.

Chapter 3. LabOne API Programming

Each of the LabOne APIs (LabVIEW, Matlab, Python, C, .NET) provide an interface to configure and acquire data from your instrument. All these programming interfaces are, however, thin application layers based on a shared core API, `ziCore`. This chapter describes the low-level command and high-level functionality provided LabOne Modules that's available in all the LabOne interfaces.

Refer to:

- [Section 3.1](#) for [An Introduction to LabOne Modules](#).
- [Section 3.2](#) for [Low-level LabOne API Commands](#).
- [Section 3.4](#) for the [AWG Module](#).
- [Section 3.5](#) for the [Data Acquisition Module](#).
- [Section 3.6](#) for the [Device Settings Module](#).
- [Section 3.7](#) for the [Impedance Module](#).
- [Section 3.8](#) for the [Multi-Device Synchronisation Module](#).
- [Section 3.9](#) for the [PID Advisor Module](#).
- [Section 3.10](#) for the [Scope Module](#).
- [Section 3.11](#) for the [Sweeper Module](#).

The following modules are still maintained, but will be made deprecated in a future release. As of LabOne 17.12 the Data Acquisition Module combines and replaces the functionality of the Software Trigger (Recorder) Module and the Spectrum (ZoomFFT) Module. New users should use the Data Acquisition Module instead of these modules.

- [Section 3.12](#) for the [Software Trigger \(Recorder\) Module](#).
- [Section 3.13](#) for the [Spectrum Analyzer Module](#).

3.1. An Introduction to LabOne Modules

All of the LabOne APIs are based on a central API called `ziCore`. This allows them to share a common structure which provides a uniform interface for programming Zurich Instruments devices. The aim of this section is to familiarize the user with the key `ziCore` programming concepts which can then be used in any of the LabOne APIs (LabVIEW, Matlab, Python, .Net, and C).

3.1.1. Software Architecture

Each of the `ziCore`-based APIs are designed to have a minimal code footprint: They are simply small interface layers that use the functionality derived from `ziCore`, a central C++ API. The derived API interfaces (LabVIEW, Matlab, Python, .NET and C) provide a familiar interface to the user and allow the user to receive and manipulate data from their instrument using the API language's native data types and formats. See [Section 1.2](#) for an overview of the LabOne software architecture.

3.1.2. `ziCore` Modules

In addition to the usual API commands available for instrument configuration and data retrieval, e.g., `setInt`, `poll`), `ziCore`-based APIs also provide a number of so-called **Modules**: high-level interfaces that perform common tasks such as sweeping data or performing [FFTs](#).

The Module's functionality is implemented in `ziCore` and each derived high-level API simply provides an interface to that module from the API's native environment. This design ensures that the user can expect the same behavior from each module irrespective of which API is being used; if the user is familiar with a module available in one high-level programming API, it is quick and easy to start using the module in a different API. In particular, the LabOne User Interface is also based on `ziCore` and as such, the user can expect the same behavior using a `ziCore`-based API that is experienced in the LabOne User Interface, see [Figure 3.1](#).

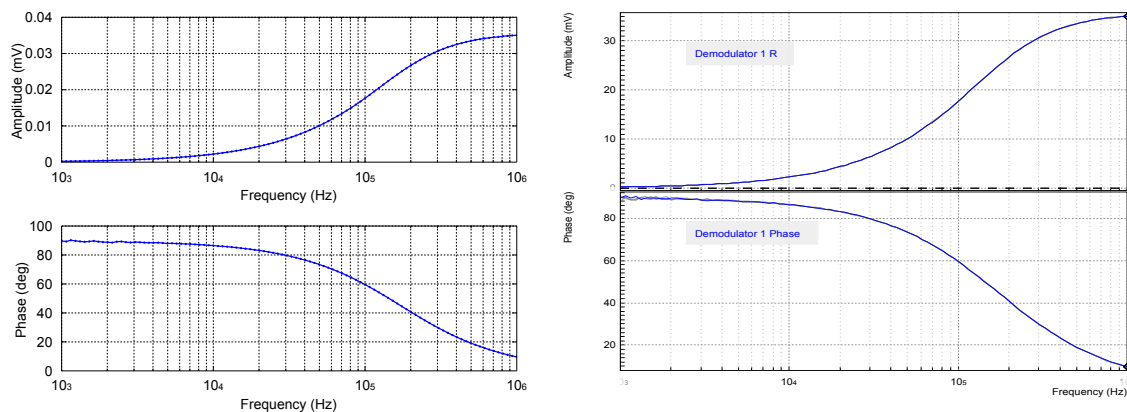


Figure 3.1. The same results and behavior can be obtained from Modules in any `ziCore`-based interface; [Sweeper Module](#) results from the LabOne Matlab API (left) and the LabOne User Interface (right) using the same Sweeper and instrument settings.

The modules currently available in `ziCore` are:

- The [Sweeper Module](#) for obtaining data whilst performing a sweep of one of the instrument's setting, e.g., measuring a frequency response.
- The [Data Acquisition Module](#) for recording instrument data [asynchronously](#) based upon user-defined triggers.

- The [Spectrum Analyzer Module](#) for calculating the FFT of demodulator output. This module is deprecated. We recommend using the [Data Acquisition Module](#) instead.
- The [Software Trigger \(Recorder\) Module](#) for recording instrument data [asynchronously](#) based upon user-defined triggers. This module is deprecated. We recommend using the [Data Acquisition Module](#) instead.
- The [Device Settings Module](#) for saving and loading instrument settings to and from (XML) files.
- The [PID Advisor Module](#) for modeling and simulating the PID incorporated in the instrument.
- The [Scope Module](#) for obtaining scope data from the instrument.
- The [Impedance Module](#) for performing impedance measurements.
- The [Multi-Device Synchronisation Module](#) for synchronizing the timestamps of multiple instruments.
- The [AWG Module](#) for working with the AWG.

In addition to providing a unified-interface between APIs, modules also provide a uniform workflow regardless of the functionality the module performs (e.g., sweeping, recording data).

An important difference to low-level `ziCore` API commands is that Modules execute their commands **asynchronously**, see [Section 3.1.3](#).

3.1.3. Synchronous versus Asynchronous Commands

The low-level API commands such as `setInt` and `poll` are **synchronous** commands, that is the interface will be blocked until that command has finished executing; the user can not run any commands in the meantime. Another feature of `ziCore`'s Modules is that each instantiation of a Module creates a new [Thread](#) and, as such, the commands executed by a Module are performed **asynchronously**. Asynchronous means that the task is performed in the background and the interface's process is available to perform other tasks in the meantime, i.e., Module commands are non-blocking for the user.

3.1.4. Converting LabOne's "systemtime" to Local Time

Data returned by Core Modules, for example the data of a single sweep, contain a header with a `systemtime` field whose value is the POSIX time in microseconds at the point in time when the data was required. It may correspond to the start of data acquisition or the end, depending on the module, but will be consistent for all objects returned from one module. In order to help convert this timestamp to an API environment's native time format there are utility functions in the LabOne APIs, where appropriate the example code in the function's docstring demonstrates their use. Please check the utility function of the respective API for more details.

3.2. Low-level LabOne API Commands

This section describes the API commands `getSample()`, `subscribe` and `poll()`. For continuous data acquisition, however, it is recommended to use the [Data Acquisition Module](#) in continuous mode.

3.2.1. The `getSample` command

For one-shot measurement demodulator data

The simplest function to obtain demodulator data is the `getSample` command. It returns a single sample from one demodulator channel, i.e., it returns the sample fields (not only the demodulator outputs X and Y) described in [Section 2.4](#) at one timestamp. The `getSample` function returns the last sample for the specified demodulator that the Data Server has received from the instrument.

The `getSample` command raises a `ZIAPITimeoutException` if no demodulator data is received from the device within 5 seconds. This is the case if the requested demodulator is not enabled. As `getSample` only returns data from a single demodulator, wildcard path specification (e.g., `/devn/demods/*/sample`) is not supported.

If multiple samples (even from one demodulator channel) are required, it is recommended to use either [subscribe and poll](#) (for high performance API applications) or [Data Acquisition Module](#). Using `getSample` in anything other than low-speed loops data is not recommended.

3.2.2. The `subscribe` and `poll` commands

For high-performance continuous or block streaming data

The `subscribe` and `poll` functions are low-level commands that allow high-speed data transfer from the instrument to the API. The idea is as following. The user may subscribe to one or more [nodes](#) in the API session by calling `subscribe` for each node. This tells the Data Server to create a buffer for each subscribed node and to start accumulating the data corresponding to the node that is streamed from the instrument (or instruments, as the case may be). The user can then call `poll` (in the same API session) to transfer the data from the Data Server's buffers to the API's client code. If `poll` is not called within 5 seconds of either subscribing, the last call to `poll` or calling the `sync` command (more information on `sync` below), the Data Server clears its buffers for the subscribed nodes and starts accumulating data again. This means that for continuous transfer of data, the user must regularly poll data from the Data Server to ensure that no data is lost in between polls.

Simple Example

For example, the following Python code subscribes to the first and fifth demodulator sample streaming nodes on a lock-in amplifier:

```
import zhinst.ziPython
daq = zhinst.ziPython('localhost', 8004, 6)
# Enable the demodulator output and set the transfer rate.
# This ensure the devices actually pushes data to the Data Server.
daq.setInt('/dev2006/demods/0/enable', 1)
daq.setInt('/dev2006/demods/4/enable', 1)
# This value will be corrected to the nearest legal value by the instrument's FW.
daq.setDouble('/dev2006/demods/0/rate', 10e3)
daq.setDouble('/dev2006/demods/4/rate', 10e3)
daq.subscribe('/dev2006/demods/0/sample')
daq.subscribe('/dev2006/demods/4/sample')
time.sleep(1) # Subscribed data is being accumulated by the Data Server.
data = daq.poll(0.020, 10, 0, True)
data.keys()
# Out: dict_keys(['/dev2006/demods/0/sample', '/dev2006/demods/4/sample'])
```



```
len(data['/dev2006/demods/0/sample']['timestamp'])
# Out: 13824
len(data['/dev2006/demods/4/sample']['timestamp'])
# Out: 13746
```

Example 3.1. Python code demonstrating basic subscribe/poll behaviour

The subscribe and poll commands return data from streaming nodes as sent from the instrument's firmware at the configured data rate. The data returned is the discrete device data as sampled or calculated by the digital signal processing algorithms on the instrument. As explained in [Section 2.2.1, Streaming Nodes](#) the data from multiple nodes may not be aligned due to different sampling rates or different streaming node sources.

When you call subscribe on a node the Data Server starts accumulating data in a dedicated buffer (for that node and the API session from which subscribe was called). When the buffer becomes full, then the data is discarded by the Data Server (poll must be called frequently enough to ensure that you do not lose data in between multiple poll calls). When you call poll, then all the data that has accumulated in the Data Server's buffers (of subscribed nodes) is returned (it is transferred to the API client) and is deleted from the Data Server's buffers. The Data Server continues to accumulate new data sent from the device for the subscribed nodes after polling and will continue to do so until unsubscribe is called for that node. It is good practice to call unsubscribe on the nodes when you no longer want to poll data from them so that the Data Server stops accumulating data and can free up the system's memory.

If you would like to continue recording new data for a subscribed node, but discard older data, then either poll can be called (and the returned older data simply discarded) or the sync command can be used. Sync clears the Data Server's buffers of subscribed data but has an additional function. Additionally, the sync command blocks (it is a synchronous command) until all set commands have been sent to the device and have taken effect. E.g., if you enable the instrument's signal output and want to ensure that you only receive data from poll after the setting has taken effect the device, then sync should be called after calling set and before calling poll.

The first two mandatory parameters to poll are the "poll duration" and the "poll timeout" parameters: Poll is also a synchronous command and will block for the specified poll duration. It will return the data accumulated during the time in seconds specified by the poll duration, and as mentioned above, it will also return the data previously accumulated data before calling poll. If you would like to poll data continuously in a loop, then a very small poll duration should be used, e.g., 0.05 seconds, so that it only blocks for this time. The poll timeout parameter must not be typically set very carefully and a value of 100 ms is sufficient. It is indeed only relevant if it set to a larger value than poll duration. In this case, if no data is arriving from the instrument, poll will wait for poll timeout milliseconds before returning. If data does arrive after poll duration but before the poll timeout, poll returns immediately. It is recommended to simply use a poll timeout of 100 ms (or if poll_duration is smaller, to set it equal to the poll_duration). Unfortunately, the units of poll duration and poll timeout differ: The poll duration is in seconds, whereas poll timeout is in milliseconds.

Ensure synchronization of settings before streaming data (sync)

Therefore, a special sync command is supported which ensures that the API is blocking during the full command execution of sending down a marker to the device and receiving it again over the API. During that time all buffers are cleaner. Therefore, after the sync command the newly recorded poll data will be later in time than the previous set commands. Be aware that this command is quite expensive ~100ms.

Asking poll to always return a value for a settings node (getAsEvent)

The poll command only returns value changes for subscribed nodes; no data will be returned for the node if it has not changed since subscribing to it. If poll should also return the value of a node that has not changed since subscribing, then `getAsEvent` may be used instead or in addition to subscribe. This ensures that a settings node value is always pushed.

```
daq = zhinst.ziPython.ziDAQServer('localhost', 8004, 6)
# Without getAsEvent no value would be returned by poll.
daq.getAsEvent('/dev2006/sigouts/0/amplitudes/3')
daq.subscribe('/dev2006/sigouts/0/amplitudes/3')
daq.poll(0.200, 10, 0, True)
# Out: {'/dev2006/sigouts/0/amplitudes/0': {'timestamp': array([26980521883432],
#      dtype=uint64),
#      'value': array([ 0.30001831])}}
```

Example 3.2. Python code demonstrating `getAsEvent` usage

If no data was stored in the Data Server's data buffer after issuing a `poll`, the command will wait for the data until the timeout time. If the buffer is empty after timeout time passed, `poll` will either simply return an empty data structure (for example, an empty dictionary in Python) or throw an error, depending which flags it has been provided with.

Note

Often one of the LabOne `ziCore` Modules provide an easier and more efficient choice for data acquisition than the comparably low-level `poll` command. Each `ziCore` Module is a software component that performs a specific high-level measurement task, for example, the [Data Acquisition Module](#) can be used to record bursts of data when a defined trigger condition is fulfilled or the [Sweeper Module](#) can be used to perform a frequency response analysis. See [Section `ziCore` Modules](#) for an overview of the available Modules.

Explanation of buffering in the Data Server and API Client

The following graphics illustrate how data are stored and transferred between the Instrument, the Data Server, and the API session in case the API session is the only client of the Data Server. [Figure 3.2](#) shows the situation when the API session has subscribed to a node, but no `poll` command is being sent. [Figure 3.3](#) corresponds to the situation when the `poll` command with a recording time of 0 is sent in regular intervals, and illustrates the moment just before the last `poll` command. [Figure 3.4](#) then illustrates the moment just after the last `poll` command.

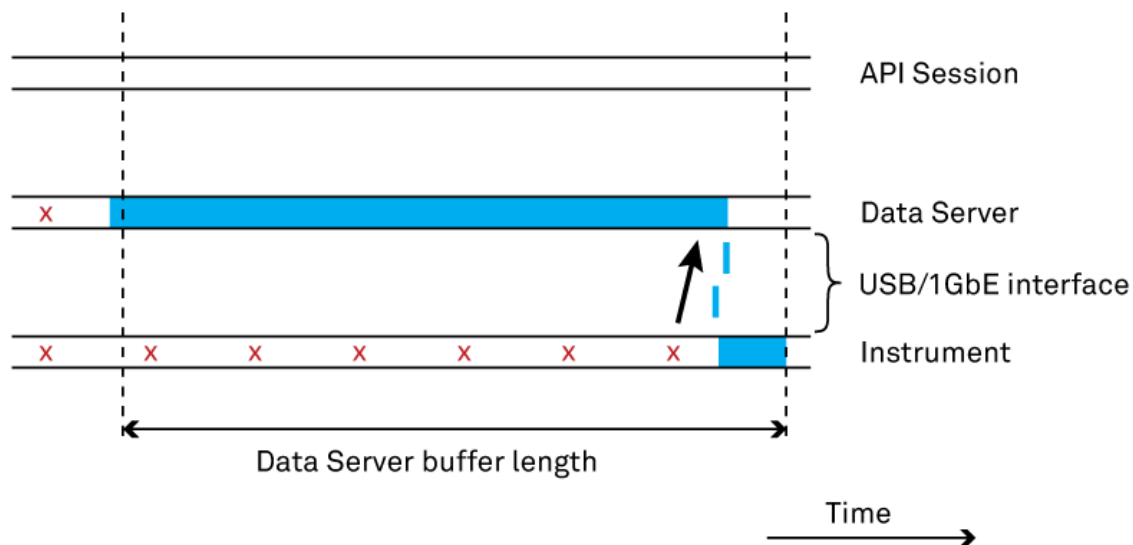


Figure 3.2. Illustration of data storage and transfer: the API Session (no other Data Server clients) is subscribed to a node (blue bars representing data stream) but never issues a `poll` command. The data are stored in the Data Server's buffer for a certain time and dumped afterwards.

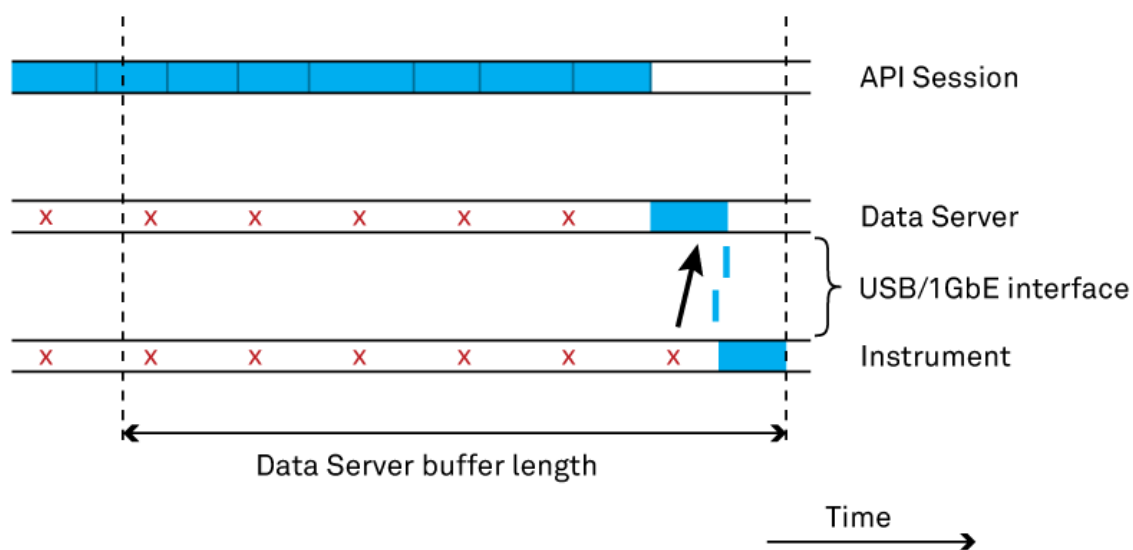


Figure 3.3. Illustration of data storage and transfer: the API Session is subscribed to a node and regularly issues a `poll` command. The Data Server holds only the data in the memory that were accumulated since the last `poll` command.

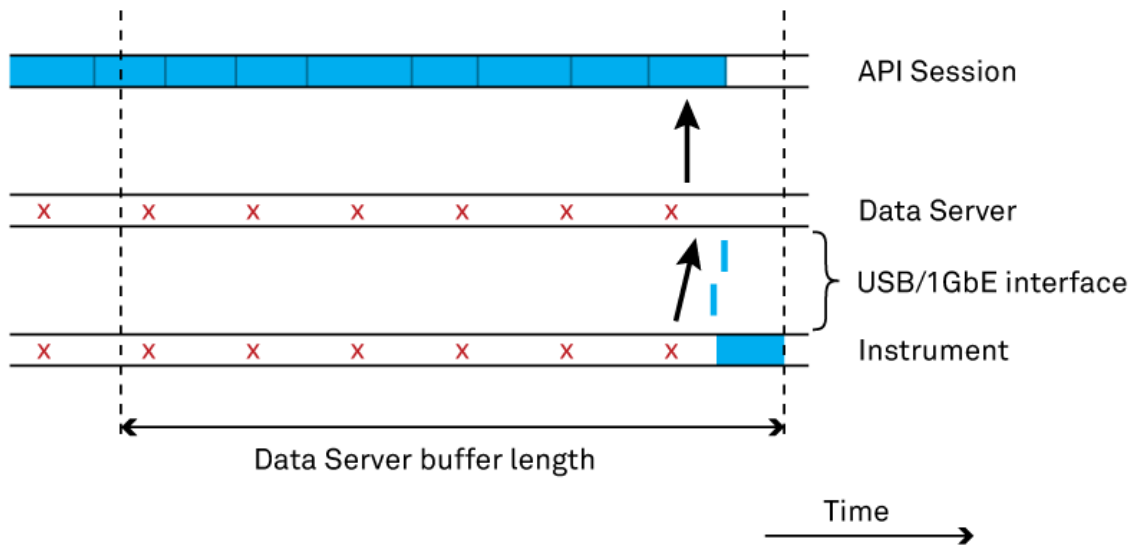


Figure 3.4. Illustration of data storage and transfer: the API Session is subscribed to a node and regularly issues a `poll` command. Upon a new `poll` command, all data accumulated in the Data Server buffer are transferred to the API Session and subsequently cleared from the Data Server buffer.

In the following cases, the picture above needs to be modified:

1. **Multiple Data Server clients:** in case multiple clients (API sessions, Web Server) are subscribed to the same node, the Data Server will keep the corresponding data in the buffer until all clients have polled the data (or until it's older than the buffer length). This means different clients will not interfere with each other.
2. **LabVIEW, C, and .NET APIs:** in these APIs (unlike in MATLAB and Python), it's not guaranteed that a single `poll` command leads to the transfer of all data in the Data Server buffer because the block size of transferred data is limited. Nonetheless, by calling `poll` frequently enough, a gapless stream of data can be obtained.
3. **HF2 Series instruments:** the buffer Data Server for HF2 Series instruments is defined by its memory size rather than by its length in units of time. This means that the duration for which the Data Server will store data depends on the sampling rate.

3.3. Common Core Module Parameters

Many of the LabOne Core Modules have the `save/*` branch of parameters which control data saving from the module, see:

- [Table 3.1](#) for input/output parameters and,
- [Table 3.2](#) for input parameters.

Table 3.1. Common Core Module `save/` branch Input/Output Parameters. The module prefix is omitted from the parameter paths; it must be included in the path when setting parameters.

Path	Type	Unit	Description
<code>save/save</code>	bool	-	Initiate the saving of data to file. The saving is done in the background. When the save has finished, the module resets this parameter to 0.

Table 3.2. Core Module `save/` branch Input Parameters. The module prefix is omitted from the parameter paths; it must be included in the path when setting parameters.

Path	Type	Unit	Description
<code>save/directory</code>	string	-	The base directory where files are saved.
<code>save/filename</code>	string	-	Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g. <code>daq_000</code> .
<code>save/fileformat</code>	string	-	The format of the file for saving data: 0: Matlab. 1: CSV. 2: SXM (Image format).
<code>save/csvseparator</code>	string	-	The character to use as CSV separator when saving files in this format.
<code>save/csvlocale</code>	string	-	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C": Dot for the decimal point and no digit grouping (default). "" (empty string): Use the symbols set in the language and region settings of the computer.
<code>save/saveonread</code>	bool	-	Automatically save the data to file immediately before reading out the data from the module using the <code>read()</code> command. Set this parameter to 1 if you want to save data to file when running the module continuously and performing intermediate reads.

3.4. AWG Module

The AWG Module allows programmers to access the functionality available in the LabOne User Interface AWG tab. It allows users to compile and upload sequencer programs to the arbitrary waveform generator on UHF and HDAWG instruments from any of the LabOne APIs.

This chapter only explains the specifics for working with an AWG from an API; reference documentation of the LabOne AWG Sequencer Programming Language can be found in the UHF or HDAWG User Manual.

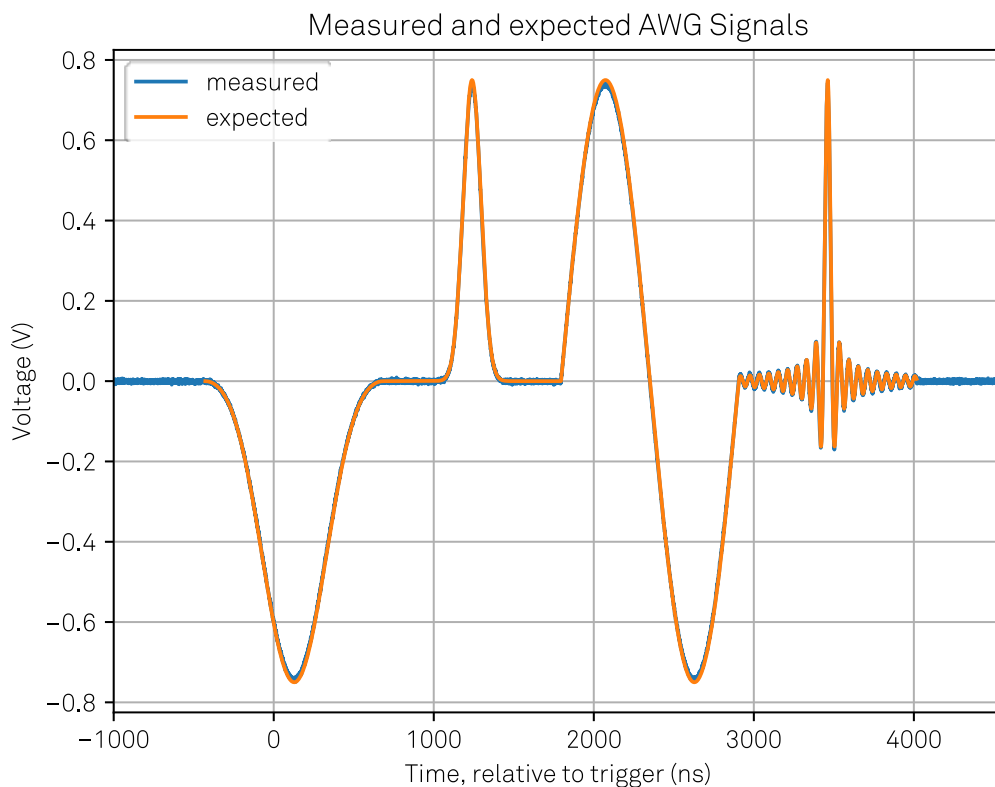


Figure 3.5. An AWG signal generated and measured using a UHFLI with the AWG Option. The waveform data was measured via a feedback cable using the UHF's Scope. The plot was generated by the Python API example for the UHF, `example_awg.py`, which also generates the expected waveform and cross-correlates it with the measured waveform in order to overlay the expected signal on the measurement data.

3.4.1. Getting Started with the AWG Module

The following API examples demonstrate AWG Module use are available:

- Matlab and Python, `example_awg.{py,m}`: Compiles and uploads an AWG sequencer program from a string. It demonstrates how to define waveforms using the four methods listed below in [Section 3.4.3](#). Separate versions of these examples are available for both UHF and HDAWG instruments.
- Matlab and Python, `example_awg_sourcefile.{py,m}`: Demonstrates how to compile and upload an AWG sequencer program from a `.seqc` file. Separate versions of these examples are available for both UHF and HDAWG instruments.

- LabVIEW `ziExample-UHFLI-Module-AWG.vi`: Compiles and uploads an AWG sequencer program from a string and captures the generated waveform in the scope (UHF only).
- .NET, `ExampleAwgModule()` (in `Examples.cs`): Compiles and uploads an AWG sequencer program from a string. It demonstrates how to define waveforms using the four methods listed below in [Section 3.4.3](#).
- C API, `ExampleAWGUpload.c`: Demonstrates how to compile and upload an AWG sequencer program from a `.seqc` file.

3.4.2. Sequencer Program Compilation and Upload

Programming an AWG with a sequencer program is a 2-step process. First, the source code must be compiled to a binary ELF file and secondly the ELF file must be uploaded from the PC to the AWG on the UHF or HDAWG instrument. Both steps are performed by the AWG Module regardless of whether the module is used in the API or the LabOne User Interface AWG Sequencer tab

Compilation

An AWG sequencer program can be provided to the AWG module for compilation as either a:

1. Source file: In this case the sequencer program file must reside in the "awg/src" sub-directory of the LabOne user directory. The filename (without full directory path) must be specified via the `awgModule/compiler/sourcefile` parameter and compilation is started when the in-out parameter `awgModule/compiler/start` is set to 1.
2. String: A sequencer program may also be sent directly to the AWG Module as a string (comprising of a valid sequencer program) without the need to create a file on disk. The string is sent to the module by writing it to the `awgModule/compiler/sourcestring` using the module `setString()` function. In this case, compilation is started automatically after writing the source string.

Upload

If the `awgModule/compiler/upload` parameter is set to 1 the ELF file is automatically uploaded to the AWG after successful compilation. Otherwise, it must be uploaded by setting the in-out parameter `awgModule/elf/upload` to 1. A running AWG must be disabled first in order to upload a new sequencer program and it must be enabled again after upload.

3.4.3. Methods to define Waveforms in Sequencer Programs

The waveforms played by an AWG sequencer program can be defined, or in the last case, modified, using the following four methods. These methods are demonstrated by the examples listed in [Section 3.4.1](#).

1. By using one of the waveform generation functions such as `sine()`, `sinc()`, `gauss()`, etc. defined in the LabOne AWG Sequencer programming language. See the UHF or HDAWG User Manual for full reference documentation.
2. By defining a waveform in floating point format in a CSV file. Such waveform CSV files must be located in the "awg/waveforms" sub-directory of the LabOne user directory (see explanation of `awgModule/directory` in [Table 3.7](#)). In order to compile a sequencer program that uses CSV-defined waveforms a comma-separated list of the CSV filenames (without extension) must be specified via the `awgModule/compiler/waveforms` parameter.
3. By defining a waveform as an array of floating point numbers within the sequencer program and playing it using the AWG Sequencer function `vect()`.

4. A waveform that has been previously defined, even as an array of zeros with, for example, `zeros(10000)`, may be replaced at program runtime with the following two steps:
 - a. Writing the index of the target waveform to replace to `/DEV.../AWGS/0/WAVEFORM/INDEX`.
 - b. Writing the new waveform vector using `vectorWrite` command to `/DEV.../AWGS/0/WAVEFORM/DATA`.

The index of the waveform to be replaced is defined as following. Let N be the total number of waveforms and $M > 0$ be the number of waveforms used in the sequencer program defined from CSV file. Then the index of the waveform to be replaced is defined as following:

- $\text{index} = 0, \dots, M-1$ for all waveforms defined from CSV alphabetically ordered by filename,
- $\text{index} = M, \dots, N-1$ in the order that the other (non-CSV defined) waveforms are defined in the sequencer program.

For the case of $M=0$ (no waveforms defined by CSV file), then the index is defined by:

- $\text{index} = 0, \dots, N-1$ in the order that the waveforms are defined in the sequencer program.

3.4.4. HDAWG Channel Grouping

This section explains how to configure the `awgModule/index` parameter and which AWGS node branch must be used for different channel grouping configurations. The channel grouping is defined by value of the the node `/DEV..../SYSTEM/AWG/CHANNELGROUPING` as following:

- 0: Use the outputs in groups of 2; each sequencer program controls 2 outputs. Each group $n=0,1,2,3$ of AWGs (respectively $n=0,1$ on HDAWG4 instruments) is configured by the `/DEV..../AWGS/n` branches. Each of these 4 groups requires its own instance of the AWG Module and `awgModule/index` should be set to $n=0,1,2,3$ for each group accordingly.
- 1: Use the outputs in groups of 4; each sequencer program controls 4 outputs. Each group $n=0,1$ of AWGs (respectively $n=0$ on HDAWG4 instruments) is configured by the `/DEV..../AWGS/0` and `/DEV..../AWGS/2` branches. Each of these two groups requires its own instance of the AWG Module and `awgModule/index` should be set to $n=0,1$ for each group accordingly. For HDAWG4 instruments, there is only one group of 4 outputs which is configured by the `/DEV..../AWGS/0` branch.
- 2: HDAWG8 devices only. Use the outputs in a single group of 8; the (single) sequencer program controls 8 outputs. There is only one group ($n=0$) of 8 AWGs which is configured by the `/DEV..../AWGS/0` branch. Only one instance of the AWG Module is required and its value of `awgModule/index` should be 0.

Table 3.3. Overview of the device nodes and the value of `awgModule/index` used in different channel grouping configurations on HDAWG8 instruments.

Value of CHANNELGROUPING	Number of Cores	AWG Core	Corresponding device AWG branch index	Value of <code>awgModule/index</code>
0	4	1	<code>/DEV..../AWGS/0</code>	0
		2	<code>/DEV..../AWGS/1</code>	1
		3	<code>/DEV..../AWGS/2</code>	2
		4	<code>/DEV..../AWGS/3</code>	3
1	2	1	<code>/DEV..../AWGS/0</code>	0
		2	<code>/DEV..../AWGS/2</code>	1
2	1	1	<code>/DEV..../AWGS/0</code>	0

Table 3.4. Overview of the device nodes and the value of `awgModule/index` used in different channel grouping configurations on HDAWG4 instruments.

Value of CHANNELGROUPING	Number of Cores	AWG Core	Corresponding device AWG branch index	Value of <code>awgModule/index</code>
0	2	1	/DEV..../AWGS/0	0
		2	/DEV..../AWGS/1	1
1	1	1	/DEV..../AWGS/0	0

3.4.5. AWG Module Parameters

The following tables provide a comprehensive list of the module's parameters, see:

- [Table 3.5](#) for input/output parameters,
- [Table 3.6](#) for input parameters and,
- [Table 3.7](#) for output parameters.

Table 3.5. AWG Module Input/Output Parameters. The `awgModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Setting/Path	Type	Unit	Description
<code>compiler/start</code>	bool	-	Set to 1 to start compiling the AWG sequencer program specified by <code>compiler/sourcefile</code> . The module sets <code>compiler/start</code> to 0 once compilation has successfully completed (or failed). If <code>compiler/upload</code> is enabled then the sequencer program will additionally be uploaded to the AWG upon after successful compilation.
<code>elf/upload</code>	bool	-	Set to 1 to start uploading the AWG sequencer program to the device. The module sets <code>elf/upload</code> to 0 once the upload has finished.

Table 3.6. AWG Module Input Parameters. The `awgModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Setting/Path	Type	Unit	Description
<code>compiler/sourcefile</code>	string	-	The filename of an AWG sequencer program file to compile and load. The file must be located in the "awg/src" sub-directory of the LabOne user directory. This directory path is provided by the value of the read-only <code>directory</code> parameter.
<code>compiler/sourcestring</code>	string	-	A string containing an AWG sequencer program may directly loaded to this parameter using the module command <code>setString</code> . This allows compilation and upload of a sequencer program without saving it to a file first. Compilation starts automatically after <code>compiler/sourcestring</code> has been set.

Setting/Path	Type	Unit	Description
compiler/upload	bool	-	Specify whether the sequencer program should be automatically uploaded to the AWG following successful compilation.
compiler/waveforms	string	-	A comma-separated list of waveform CSV files to be used by the AWG sequencer program.
device	string	-	The target device for AWG sequencer programs upload, e.g. 'dev2006'.
elf/file	string	-	The filename of the compiled binary ELF file. If not set, the name is automatically set based on the source filename. The ELF file will be saved by the AWG Module in the "awg/elf" sub-directory of the LabOne user directory. This directory path is provided by the value of the read-only directory parameter.
index	bool	-	The index of the current AWG Module to use when running with multiple AWG groups, see Section 3.4.4 for further explanation.
mds/enable	bool	-	Enables grouping of multiple synchronized instruments for AWG playback.
mds/group	int	-	The MDS group (<code>multiDeviceSyncModule/group</code>) to use for synchronized AWG playback.

Table 3.7. AWG Module Output (read-only) Parameters. The `awgModule/` prefix is omitted from the parameter paths.

Setting/Path	Type	Unit	Description
compiler/status	int	-	Compilation status: 1: Idle. 0: Compilation successful. 1: Compilation failed. 2: Compilation completed with warnings.
compiler/statusstring	string	-	Status message of the compiler.
directory	string	-	The path of the LabOne user directory (this may not be modified). The AWG Module uses the following subdirectories in the LabOne user directory: awg/src Contains AWG sequencer program source files (user created). awg/elf Contains compiled AWG binary (ELF) files (created by the module). awg/waves Contains CSV waveform files (user created).

Setting/Path	Type	Unit	Description
elf/status	int	-	Status of the ELF file upload: 1: Idle. 0: Upload successful. 1: Upload failed. 2: Upload in progress.
elf/checksum	int	-	The checksum of the generated ELF file.
progress	double	-	Reports the progress of the upload as a value between 0 and 1.

3.5. Data Acquisition Module

The Data Acquisition Module corresponds to the Data Acquisition tab of the LabOne User Interface. It enables the user to record and align time and frequency domain data from multiple instrument signal sources at a defined data rate. The data may be recorded either continuously or in bursts based upon trigger criteria analogous to the functionality provided by laboratory oscilloscopes.

The Data Acquisition Module returned 10 segments of demodulator data each with a duration of 0.180 seconds

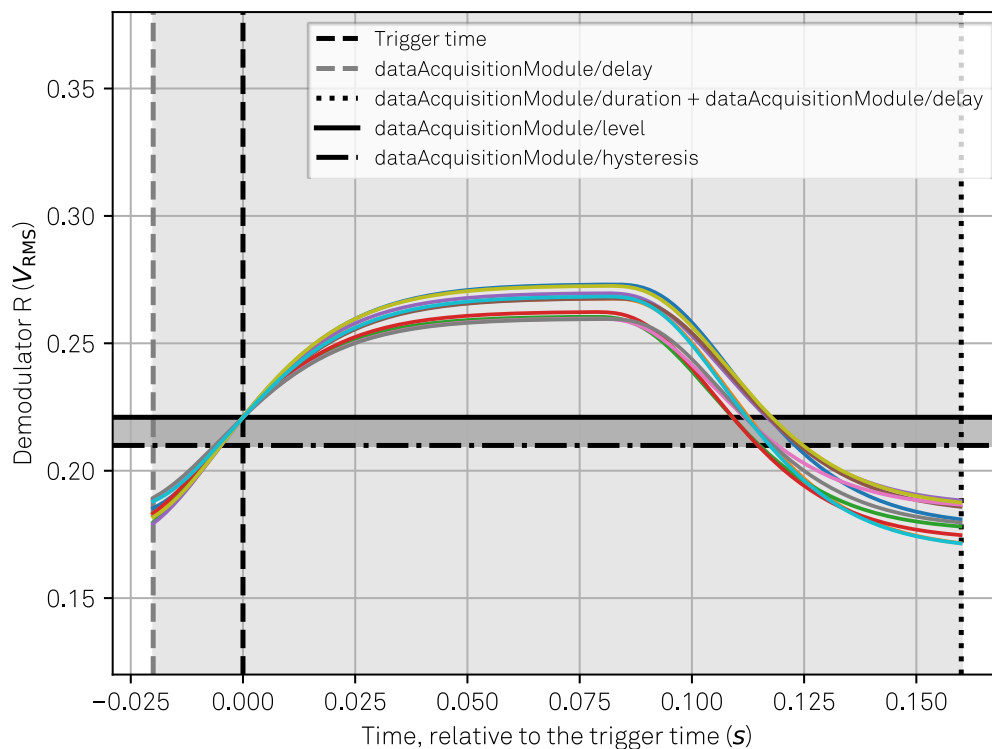


Figure 3.6. The plot produced by `example_data_acquisition_edge.py`, an example distributed with the LabOne Python API. The plot shows 10 bursts of data acquired from a demodulator; each burst was recorded when the demodulator's R value exceeded a specified threshold using a positive edge trigger. See [Section 5.2.3](#) for help getting started with the Python examples.

Historical Note

The Data Acquisition Module was introduced in LabOne Release 17.12 and improved and unified the functionality of the [Software Trigger \(Recorder\) Module](#) and the [Spectrum Analyzer Module](#). The DAQ and Spectrum tabs in the LabOne User Interface both use the Data Acquisition Module. The main new features of the DAQ Module (in comparison to the Software Trigger Module) are:

- Simultaneous triggered time and frequency domain (FFT) data acquisition.
- Exact mode, in addition to linear or nearest neighbour interpolation sampling modes. Exact mode ensures that all the subscribed data is sampled onto the grid defined by the subscribed signal with the highest data rate.
- A continuous acquisition mode.

- Dot notation for specifying the exact signal to acquire and any operations to be performed on the data, e.g. averaging, standard deviation, FFT.

Please note that the [Software Trigger \(Recorder\) Module](#) and the [Spectrum Analyzer Module](#) will be deprecated in a future release. [Section 3.5.4](#) and [Section 3.5.5](#) explain how to migrate existing Software Trigger, respectively zoomFFT, API programs to the Data Acquisition Module.

3.5.1. DAQ Module Acquisition Modes and Trigger Types

This section lists the required parameters and special considerations for each trigger mode. For reference documentation of the module's parameters please see [Section 3.5.6](#).

Table 3.8. Overview of the acquisition modes available in the Data Acquisition Module.

Mode / Trigger Type	Description	Value of <code>dataAcquisitionModule/type</code>
Continuous	Continuous recording of data.	0
Edge	Edge trigger with noise rejection.	1
Pulse	Pulse width trigger with noise rejection.	3
Tracking (Edge or Pulse)	Level tracking trigger to compensate for signal drift.	4
Digital	Digital trigger with bit masking.	2
Hardware	Trigger on one of the instrument's hardware trigger channels (not available on HF2).	6
Pulse Counter	Trigger on the value of an instrument's pulse counter (requires CNT Option).	8

Continuous Acquisition

This mode performs back-to-back recording of the subscribed signal paths. The data is returned by `read()` in bursts of a defined length (`dataAcquisitionModule/duration`). This length is defined either:

- Directly by the user via `dataAcquisitionModule/duration` for the case of nearest or linear sampling (specified by `dataAcquisitionModule/grid/mode`).
- Set by the module in the case of exact grid mode based on the value of `dataAcquisitionModule/grid/cols` and the highest sampling rate rate of all subscribed signal paths.

Acquisition using Level Edge Triggering

Parameters specific to edge triggering are:

- `dataAcquisitionModule/level`,
- `dataAcquisitionModule/hysteresis`.

The user can request automatic calculation of the `level` and `hysteresis` parameters by setting the `findlevel` parameter to 1. Please see [Section 3.5.2](#) for more information.

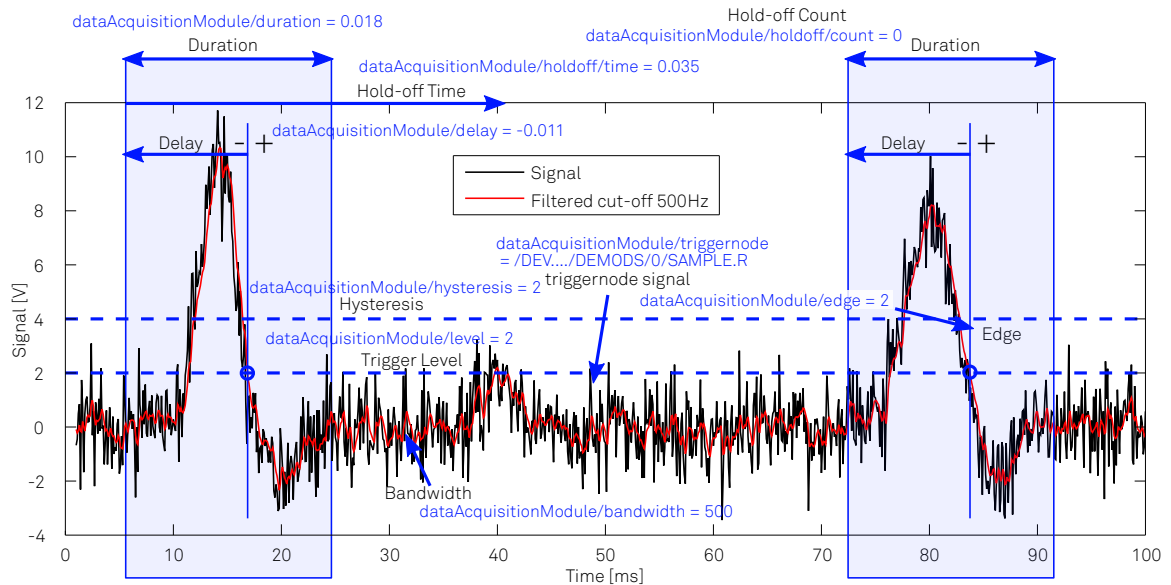


Figure 3.7. Explanation of the Data Acquisition Module's parameters for an Edge Trigger.

Acquisition using Pulse Triggering

Parameters specific to pulse triggering are:

- `dataAcquisitionModule/level`,
- `dataAcquisitionModule/hysteresis`,
- `dataAcquisitionModule/pulse/min`,
- `dataAcquisitionModule/pulse/max`.

The user can request automatic calculation of the `level` and `hysteresis` parameters by setting the `findlevel` parameter to 1. Please see [Section 3.5.2](#) for more information.

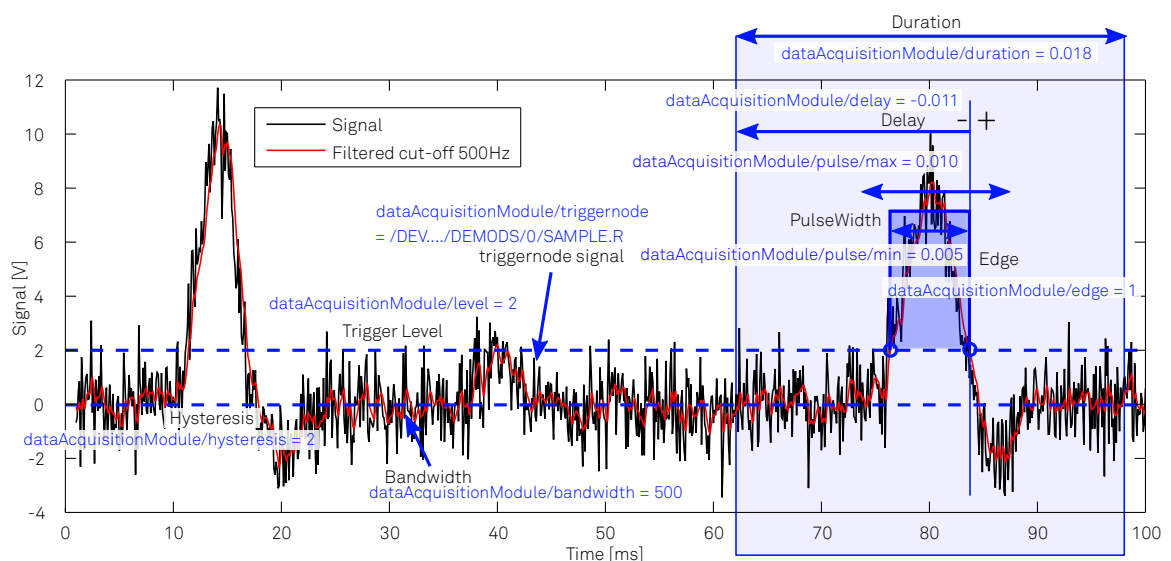


Figure 3.8. Explanation of the Data Acquisition Module's parameters for a positive Pulse Trigger.

Acquisition using Tracking Edge or Pulse Triggering

In addition to the parameters specific to edge and pulse triggers, the parameter that is of particular importance when using a tracking trigger type is:

- `dataAcquisitionModule/bandwidth`,

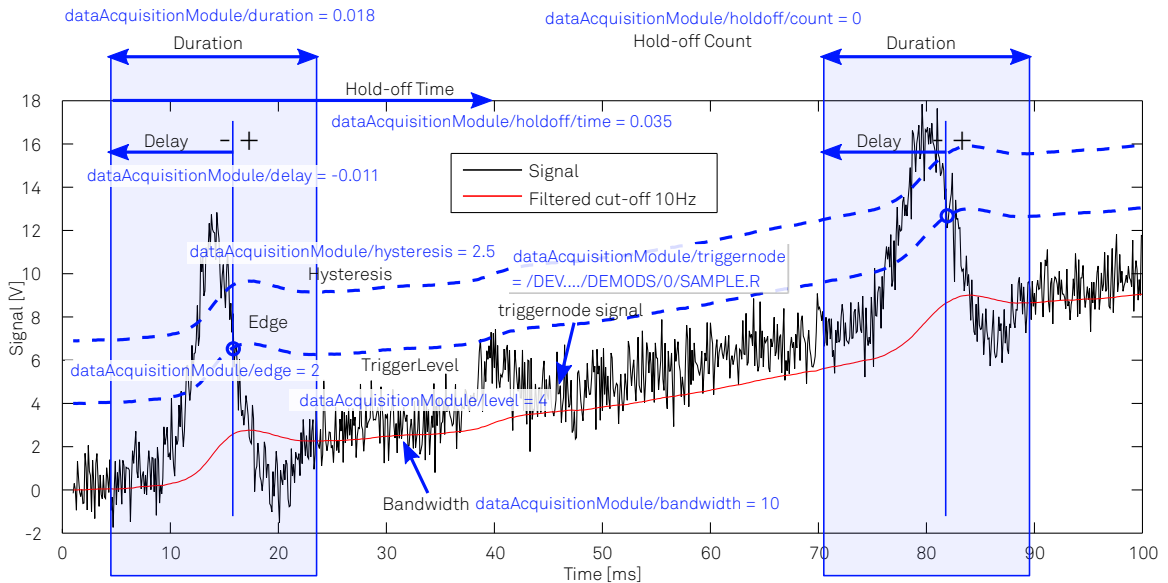


Figure 3.9. Explanation of the Data Acquisition Module's parameters for a Tracking Trigger.

Acquisition using Digital Triggering

To use the DAQ Module with a digital trigger, it must be configured to use a digital trigger type (by setting `dataAcquisitionModule/type` to 2) and to use the output value of the instrument's DIO port as its trigger source. This is achieved by setting `dataAcquisitionModule/triggernode` to the device node `/DEV.../DEMOS/N/SAMPLE.bits`). It is important to be aware that the Data Acquisition Module takes its value for the DIO output from the demodulator sample field `bits`, not from a node in the `/DEV.../DIOS/` branch. As such, the specified demodulator must be enabled and an appropriate transfer rate configured that meets the required trigger resolution (the Data Acquisition Module can only resolve triggers at the resolution of $1/(\text{DEV.../DEMOS/N/RATE})$; it is not possible to interpolate a digital signal to improve trigger resolution and if the incoming trigger pulse on the DIO port is shorter than this resolution, it may be missed).

The Digital Trigger allows not only the trigger bits (`dataAcquisitionModule/bits`) to be specified but also a bit mask (`dataAcquisitionModule/bitmask`) in order to allow an arbitrary selection of DIO pins to supply the trigger signal. When a positive, respectively, negative edge trigger is used, all of these selected pins must become high, respectively low. The bit mask is applied as following. For positive edge triggering (`dataAcquisitionModule/edge` set to value 1), the Data Acquisition Module recording is triggered when the following equality holds for the DIO value:

```
(/DEV.../DEMOS/N/SAMPLE.bits BITAND dataAcquisitionModule/bitmask) ==
(dataAcquisitionModule/bits BITAND dataAcquisitionModule/bitmask)
```

and this equality has not been met for the previous value in time (the previous sample) of `/DEV.../DEMOS/N/SAMPLE.bits`. For negative edge triggering (`dataAcquisitionModule/edge` set to value 2), the Data Acquisition Module recording is triggered when the following inequality holds for the current DIO value:

```
(/DEV..../DEMODS/N/SAMPLE.bits BITAND dataAcquisitionModule/bitmask) !=  
(dataAcquisitionModule/bits BITAND dataAcquisitionModule/bitmask)
```

and this inequality was not met (there was equality) for the previous value of the DIO value.

Acquisition using Hardware Triggering

There are no parameters specific only to hardware triggering since the hardware trigger defines the trigger criterion itself; only the trigger edge must be specified. For a hardware trigger the `dataAcquisitionModule/triggernode` must be one of:

- `/DEV.../CNTS/N/SAMPLE.TrigAWGTrigN` (requires CNT Option),
- `/DEV.../DEMODS/N/SAMPLE.TrigAWGTrigN`,
- `/DEV.../DEMODS/N/SAMPLE.TrigDemod4Phase`,
- `/DEV.../DEMODS/N/SAMPLE.TrigDemod8Phase`,
- `/DEV.../CNTS/N/SAMPLE.TrigInN` (requires CNT Option),
- `/DEV.../DEMODS/N/SAMPLE.TrigInN`,
- `/DEV.../DEMODS/N/SAMPLE.TrigOutN`.

The hardware trigger type is not supported on HF2 instruments.

Acquisition using Pulse Counter Triggering

Pulse Counter triggering requires the CNT Option. Parameters specific to the pulse counter trigger type:

- `eventcount/mode`.

The `dataAcquisitionModule/triggernode` must be configured to be a pulse counter sample:

- `/DEV..../CNTS/N/SAMPLE.value`

3.5.2. Determining the Trigger Level automatically

The Data Acquisition Module can calculate the `dataAcquisitionModule/level` and `dataAcquisitionModule/hysteresis` parameters based on the current input signal for edge, pulse, tracking edge and tracking pulse trigger types. This is particularly useful when using a tracking trigger, where the trigger level is relative to the output of the low-pass filter tracking the input signal's average (see [Figure 3.9](#)). In the LabOne User Interface this functionality corresponds to the "Find" button in the Settings sub-tab of the Data Acquisition Tab.

This functionality is activated via API by setting the `dataAcquisitionModule/findlevel` parameter to 1. This is a single-shot calculation of the level and hysteresis parameters, meaning that it is performed only once, not continually. The Data Acquisition Module monitors the input signal for a duration of 0.1 seconds and sets the level parameter to the average of the largest and the smallest values detected in the signal and the hysteresis to 10% of the difference between largest and smallest values. When the Data Acquisition Module has finished its calculation of the level and hysteresis parameters it sets the value of the `dataAcquisitionModule/findlevel` parameter to 0 and writes the values to the `dataAcquisitionModule/level` and `dataAcquisitionModule/hysteresis` parameters. Note that the calculation is only

performed if the Data Acquisition Module is currently running, i.e., after `execute()` has been called. See [Example 3.4](#) for Python code demonstrating how to use this behaviour.

```
# Arm the Data Acquisition Module: ready for trigger acquisition.
trigger.execute()
# Tell the Data Acquisition Module to determine the trigger level.
trigger.set('dataAcquisitionModule/findlevel', 1)
findlevel = 1
timeout = 10 # [s]
t0 = time.time()
while findlevel == 1:
    time.sleep(0.05)
    findlevel = trigger.getInt('dataAcquisitionModule/findlevel')
    if time.time() - t0 > timeout:
        trigger.finish()
        trigger.clear()
        raise RuntimeError("Data Acquisition Module didn't find trigger level after
        %.3f seconds." % timeout)
level = trigger.getDouble('dataAcquisitionModule/level')
hysteresis = trigger.getDouble('dataAcquisitionModule/hysteresis')
```

Example 3.3. Python code demonstrating how to use the `dataAcquisitionModule/findlevel` parameter. Taken from the Python example `example_data_acquisition_grid`.

3.5.3. Signal Subscription

The Data Acquisition Module uses dot notation for subscribing to the signals. Whereas with the Software Trigger (Recorder Module) you subscribe to an entire streaming node, e.g. `/DEV.../DEMOS/N/SAMPLE` and get all the signal components of this node back, with the Data Acquisition Module you specify the exact signal you are interested in capturing, e.g. `/DEV.../DEMOS/N/SAMPLE.r`, `/DEV.../DEMOS/0/SAMPLE.phase`.

In addition, by appending suffixes to the signal path, various operations can be applied to the source signal and cascaded to obtain the desired result. Some examples are given below (the `/DEV.../DEMOS/n/SAMPLE` prefix has been omitted):

<code>x</code>	Demod sample <code>x</code> component.
<code>r.avg</code>	Average of demod sample <code>abs(x + iy)</code> .
<code>x.std</code>	Standard deviation of demod sample <code>x</code> component.
<code>xiy.fft.abs.std</code>	Standard deviation of complex FFT of <code>x + iy</code> .
<code>phase.fft.abs.avg</code>	Average of real FFT of linear corrected phase.
<code>freq.fft.abs.pwr</code>	Power of real FFT of frequency.
<code>r.fft.abs</code>	Real FFT of <code>abs(x + iy)</code> .
<code>df.fft.abs</code>	Real FFT of demodulator phase derivative $((d\theta/dt)/(2\pi))$.
<code>xiy.fft.abs.pwr</code>	Power of complex FFT of <code>x + iy</code> .
<code>xiy.fft.abs.filter</code>	Demodulator low-pass filter transfer function. Divide <code>xiy.fft.abs</code> by this to obtain a compensated FFT.

The specification for signal subscription is given below together with the possible options. Angle brackets `<>` indicate mandatory fields. Square brackets `[]` indicate optional fields.

```
<node_path><.source_signal>[.fft<.complex_selector>[.filter]] [.pwr]
[.math_operation]
```

Table 3.9. Signal Subscription Options

Name		Description	Comment
node_path		Path of the node containing the signal(s)	
source_signal			
Node	Signal Name	Description	Comment
demod	x	Demodulator output in-phase component	
	y	Demodulator output quadrature component	
	r	Demodulator output amplitude	
	theta	Demodulator output phase	
	frequency	Oscillator frequency	
	auxin0	Auxilliary input channel 1	
	auxin1	Auxilliary input channel 2	
	xiy	Combined demodulator output in-phase and quadrature components	complex output (can only be used as FFT input))
impedance	df	Demodulator output phase derivative (can only be used for $\text{FFT}(d\theta/dt)/(2\pi)$)	
	realz	In-phase component of impedance sample	
	imagz	Quadrature component of impedance sample	
	absz	Amplitude of impedance sample	
	phasez	Phase of impedance sample	
	frequency	Oscillator frequency	
	param0	Measurement parameter that depends on circuit configuration	
	param1	Measurement parameter that depends on circuit configuration	
	drive	Amplitude of the AC signal applied to the device under test	
	bias	DC Voltage applied to the device under test	
	z	Combined impedance in-phase and quadrature components	complex (can only be used as FFT input)
other		Nodes not listed here	Nodes containing only one signal do not have a source_signal field.
FFT (optional)			
Name		Description	Comment
FFT			complex output

complex_selector (mandatory with FFT)		
Name	Description	Comment
real	Real component of FFT	
imag	Imaginary component of FFT	
abs	Absolute component of FFT	
phase	Phase component of FFT	
filter (optional)		
Name	Description	Comment
filter	Helper signal representing demodulator low-pass filter transfer function. It can only be applied to 'abs' FFT output of complex demodulator source signal, i.e. 'xiy.fft.abs.filter'. No additional operations are permitted. Can be used to compensate the FFT result for the demodulator low-pass filter.	
pwr (optional)		
Name	Description	Comment
pwr	Power calculation	
math_operation (optional)		
Name	Description	Comment
avg	Average of grid repetitions (parameter grid/repetitions)	
std	Standard deviation	

3.5.4. Migrating a SW Trigger program to the DAQ Module

We strongly recommend updating API programs that use the Software Trigger Module to use the Data Acquisition Module since the former may no longer be supported in future releases. This section provides a guide on which parameters to change in order to achieve this.

- Replace the instantiation of the Software Trigger Module with the Data Acquisition Module ('dataAcquisitionModule').
- Replace "trigger/0" with "dataAcquisitionModule" in parameter paths.
- Replace "trigger/" with "dataAcquisitionModule/" in all parameter paths.
- Decide on which grid mode to use ("none" is no longer available).
- If exact grid mode is selected, set grid/cols to an appropriate value, i.e. the number of samples to capture for each trigger. The duration then automatically becomes ("grid/cols") / sampling rate (of the signal with the highest sampling rate when more than one is subscribed).
- If using a non-exact mode (either nearest or linear interpolation), set the duration to the desired value.

- Subscribe individually to the signals to be captured using dot notation the same way as setting the trigger node, e.g. `"/DEV..../DEMOS/N/SAMPLE.r"`. If averaging is activated (`grid/repetitions > 1`), subscribe using the `.avg` suffix, e.g. `"/DEV..../DEMOS/N/SAMPLE.r.avg"`.
- The module can now be started by setting the `enable` parameter to 1 instead of calling `execute()`.
- Remove any setting of the parameter `dataAcquisitionModule/buffersize`, this is automatically set by the module and is now read-only.
- Access the data read from the module in the normal way but note that in the case of demodulator data, the signal is part of the path, e.g. `"/DEV..../DEMOS/N/SAMPLE.r"`. Note also that in the case of MATLAB, the dot is replaced by underscore, e.g. `"/DEV..../DEMOS/N/SAMPLE_r"`, to prevent conflicts with the syntax for accessing fields.
- Note that the retrigger feature that was in the Software Trigger Module is not supported in the Data Acquisition Module.

An important point to remember is that the DAQ Module always returns a grid of samples given by the parameters `dataAcquisitionModule/grid/cols` and `dataAcquisitionModule/grid/rows`. In exact grid mode, the trigger duration is given by the subscribed signal with the highest sampling rate. Signals with lower sampling rates are up-sampled using linear interpolation to give the required number of grid samples.

3.5.5. Migrating a zoomFFT program to the DAQ Module

We strongly recommend updating API programs that use the old zoomFFT Module to use the Data Acquisition Module since the former may no longer be supported in future releases. This section provides a guide on which parameters to change in order to achieve this.

- Replace the instantiation of the zoomFFT Module with Data Acquisition Module (`'dataAcquisitionModule'`).
- Replace `"zoomFFT/"` with `"dataAcquisitionModule/"` in all parameter paths.
- Remove any references to `zoomFFT/bit`.
- Set `"dataAcquisitionModule/grid/cols"` to a binary power to set the number of samples.
- Set the `"dataAcquisitionModule/type"` to 0 for untriggered FFTs. Alternatively specify a different value to enable triggered FFTs.
- Subscribe individually to the signals on which FFTs are to be performed and add the `".fft"` operator followed by the complex operator (`".real"`, `".imag"`, `".abs"`, `".phase"`), e.g. `"/DEV..../DEMOS/N/SAMPLE.xiy.fft.abs"`. If averaging is activated (`grid/repetitions > 1`), subscribe using the `.avg` suffix, e.g. `"/DEV..../DEMOS/N/SAMPLE.xiy.fft.abs.avg"`.
- If overlapped FFTs are enabled, remove any setting of parameter `dataAcquisitionModule/overlap` (this used to set the proportion of overlap between 0.0 and 1.0). Instead set the parameter `dataAcquisitionModule/spectrum/overlapped` to 1 and set the parameter `dataAcquisitionModule/refreshrate` value (Hz) to give the rate of overlap.
- Set the parameter `dataAcquisitionModule/preview` to 1 to get lower resolution previews for longer FFTs.
- Note that the Data Acquisition Module has many more features for FFT compared to the zoomFFT module, such as grid operation, averaging, waterfall, overwrite, triggered FFTs, which can also be activated. Note also that time-domain and FFT data can be returned from the module simultaneously (i.e. the data used for the FFT is the returned time-domain data) by subscribing the the appropriate signals. There is also a greater selection of signals on which an FFT can be performed, e.g. PID, AuxIn, Boxcar, AU, Pulse Counter, DIO.

3.5.6. Data Acquisition Module Parameters

The following tables provide a comprehensive list of the module's parameters, see:

- [Table 3.10](#) for input/output parameters,
- [Table 3.11](#) for input parameters and,
- [Table 3.12](#) for output parameters.

Table 3.10. Data Acquisition Module In/Out Parameters. The `dataAcquisitionModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Setting/Path	Type	Unit	Description
<code>clearhistory</code>	bool	-	Set to 1 to clear all the acquired data from the module. The module immediately resets <code>clearhistory</code> to 0 after it has been set to 1.
<code>duration</code>	double	s	The recording length of each trigger event. This is an input parameter when the sampling mode (<code>grid/mode</code>) is either nearest or linear interpolation. In exact sampling mode <code>duration</code> is an output parameter; it is calculated and set by the module based on the value of <code>grid/cols</code> and the highest rate of all the subscribed signal paths.
<code>findlevel</code>	bool	-	Set to 1 to automatically find appropriate values of the trigger level and hysteresis based on the current <code>triggernode</code> signal value. The module sets <code>findlevel</code> to 0 once the values have been found and set. See Section 3.5.2 for further explanation.
<code>forcetrigger</code>	bool	-	Set to 1 to force acquisition of a single trigger for all subscribed signal paths (when running in a triggered acquisition mode). The module immediately resets <code>forcetrigger</code> to 0 after it has been set to 1.
<code>spectrum/autobandwidth</code>	bool	-	Set to 1 to initiate automatic adjustment of the subscribed demodulator bandwidths to obtain optimal alias rejection for the selected frequency span which is equivalent to the sampling rate. The FFT mode has to be enabled (<code>spectrum/enable</code>) and the module has to be running for this function to take effect. The module resets <code>spectrum/autobandwidth</code> to 0 when the adjustment has finished.

Table 3.11. Data Acquisition Module Input Parameters. The `dataAcquisitionModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Setting/Path	Type	Unit	Description
<code>awgcontrol</code>	bool	-	Enable interaction with the AWG. If enabled, the row number is identified based on the digital row ID number set by the AWG. If disabled, every new trigger event is attributed to a new row sequentially.
<code>bandwidth</code>	double	Hz	Set to a value other than 0 in order to apply a low-pass filter with the specified bandwidth to the <code>triggernode</code> signal before applying the trigger criteria. For edge and pulse trigger use a bandwidth larger than the trigger signal's sampling rate divided by 20 to keep the phase delay. For tracking filter use a bandwidth smaller than the trigger signal's sampling rate divided by 100 to track slow signal components like drifts. The value of the filtered signal is returned by <code>read()</code> under the path <code>/DEV..../TRIGGER/LOWPASS</code> .
<code>bitmask</code>	int	-	Specify a bit mask for the DIO trigger value. The trigger value is bits AND bit mask (bitwise).. Only used when the trigger type is digital. See Acquisition using Digital Triggering for more information.
<code>bits</code>	int	-	Specify the value of the DIO to trigger on. All specified bits have to be set in order to trigger. Only used when the trigger type is digital. See Acquisition using Digital Triggering for more information.
<code>count</code>	int	-	The number of trigger events to acquire in single-shot mode (when <code>endless</code> is set to 0).
<code>delay</code>	double	s	Time delay of trigger frame position (left side) relative to the trigger edge. delay = 0: Trigger edge at left border. delay < 0: Trigger edge inside trigger frame (pre-trigger). delay > 0 Trigger edge before trigger frame (post-trigger).
<code>device</code>	string	-	The device serial to be used with the Data Acquisition Module, e.g. <code>dev123</code> (compulsory parameter).
<code>edge</code>	int	-	The trigger edge to trigger upon when running an triggered acquisition mode: 1: Rising edge 2: Falling edge 3: Both rising and falling

Setting/Path	Type	Unit	Description
<code>enable</code>	bool	-	Set to 1 to enable the module and start data acquisition (is equivalent to calling <code>execute()</code>).
<code>endless</code>	bool	-	Set to 1 to enable endless triggering. Set to 0 and use <code>count</code> if the module should only acquire a certain number of trigger events.
<code>eventcount/mode</code>	int	-	Specifies the trigger mode when the <code>triggernode</code> is configured as a pulse counter sample value (/DEV..../CNTS/0/SAMPLE.value): 0: Trigger on every sample from the pulse counter, regardless of the counter value. 1: Trigger on incrementing counter values.
<code>fft/absolute</code>	bool	-	Set to 1 to shift the frequencies in the FFT result so that the center frequency becomes the demodulation frequency rather than 0 Hz (when disabled).
<code>fft/window</code>	int	-	The FFT window function to use (default 1 = Hann). Depending on the application it makes a huge difference which of the provided window function is used. Please check the literature to find out the best trade off for your needs. 0: Rectangular 1: Hann 2: Hamming 3: Blackman Harris 4 term 16: Exponential (ring-down) 17: Cosine (ring-down) 18: Cosine squared (ring-down)
<code>flags</code>	int	-	Record flags. FILL = 0x0001 Fill data loss holes (this flag is always enabled). ALIGN = 0x0002 Align data from multiple subscribed signals (this flag is always enabled). THROW = 0x0004 Throw an exception if sample loss is detected. DETECT = 0x0008 Just detect data loss holes (this flag is always enabled).
<code>grid/cols</code>	int	-	Specify the number of columns in the returned data grid (matrix). The data along

Setting/Path	Type	Unit	Description
			the horizontal axis is resampled to the number of samples defined by <code>grid/cols</code> . The <code>grid/mode</code> parameter specifies how the data is sample onto the time, respectively frequency, <code>grid</code> .
<code>grid/direction</code>	int	-	The direction to organize data in the grid's matrix: 0: Forward. The data in each row is ordered chronologically, e.g., the first data point in each row corresponds to the first timestamp in the trigger data. 1: Reverse. The data in each row is ordered reverse chronologically, e.g., the first data point in each row corresponds to the last timestamp in the trigger data. 2: Bidirectional. The ordering of the data alternates between Forward and Backward ordering from row-to-row. The first row is Forward ordered.
<code>grid/mode</code>	int	-	Specify how the acquired data is sample onto the matrix's horizontal axis (time or frequency). Each trigger event becomes a row in the matrix and each trigger event's subscribed data is sampled onto the grid defined by the number of columns (<code>grid/cols</code>) and resampled as following: 1: Use the closest data point (nearest neighbour interpolation). 2: Use linear interpolation. 4: Do not resample the data from the subscribed signal path(s) with the highest sampling rate; the horizontal axis data points are determined from the sampling rate and the value of <code>grid/cols</code>. Subscribed signals with a lower sampling rate are upsampled onto this grid using linear interpolation.
<code>grid/overwrite</code>	bool	-	If enabled, the module will return only one data chunk (<code>grid</code>) when it is running, which will then be overwritten by subsequent trigger events.
<code>grid/repetitions</code>	int	-	Number of statistical operations performed per grid. Only applied when the subscribed signal path is, for example, an average or a standard deviation.
<code>grid/rowrepetition</code>	int	-	Enable row-wise repetition. With row-wise repetition, each row is calculated from successive repetitions before starting the next row. With grid-wise repetition, the entire grid is calculated with each repetition.

Setting/Path	Type	Unit	Description
grid/rows	int	-	Specify the number of rows in the grid's matrix. Each row is the data recorded from one trigger event.
grid/waterfall	int	-	Set to 1 to enable waterfall mode: Move the data upwards upon each trigger event; the data from newest trigger event is placed in row 0.
historylength	bool	-	Sets an upper limit for the number of data captures stored in the module.
holdoff/count	int	-	The number of skipped trigger events until the next trigger event is acquired.
holdoff/time	double	s	The hold-off time before trigger acquisition is re-armed again. A hold-off time smaller than the duration will produce overlapped trigger frames.
hysteresis	double	many	If non-zero, <code>hysteresis</code> specifies an additional trigger criteria to <code>level</code> in the trigger condition. The trigger signal must first go higher, respectively lower, than the hysteresis value and then the trigger level for positive, respectively negative edge triggers. The hysteresis value is applied below the trigger level for positive trigger edge selection. It is applied above for negative trigger edge selection, and on both sides for triggering on both edges. A non-zero hysteresis value is helpful to trigger on the correct edge in the presence of noise to avoid false positives.
level	double	many	The trigger level value.
preview	bool	-	If set to 1, enable the data of an incomplete trigger to be read. Useful for long trigger durations (or FFTs) by providing access to the intermediate data.
refreshrate	double	Hz	Set the maximum refresh rate of updated data in the returned grid. The actual refresh rate depends on other factors such as the hold-off time and duration.
pulse/max	double	s	The maximum pulse width to trigger on when using a pulse trigger.
pulse/min	double	s	The minimum pulse width to trigger on when using a pulse trigger.
save/*	-	-	Core module save/ branch parameters, see Table 3.1 and Table 3.2 for a description of the parameters.
spectrum/enable	bool	-	Enables the FFT mode of the data Acquisition module, in addition to time domain data acquisition. Note that when the FFT mode is enabled, the <code>grid/cols</code> parameter value is rounded down to the nearest binary power.

Setting/Path	Type	Unit	Description
spectrum/frequencyspan	double	-	Sets the desired frequency span of the FFT.
spectrum/overlapped	bool	-	Enables overlapping FFTs. If disabled (0), FFTs are performed on distinct abutting data sets. If enabled, the data sets of successive FFTs overlap based on the defined refresh rate.
triggernode	string	-	The node path and signal that should be used for triggering, the node path and signal should be separated by a dot (.), e.g. /DEV.../ DEMODS/0/SAMPLE.X.
type	int	-	Specifies when the module should acquire data: 0: Continuous acquisition. 1: Level edge trigger. 2: Digital trigger mode (on DIO source). 3: Pulse trigger. 4: Level tracking trigger, see also bandwidth.. 6: Hardware trigger (on trigger line source). 7: Pulse tracking trigger, see also bandwidth.. 8: Event count trigger (on pulse counter source).

Table 3.12. Data Acquisition Module Output (read-only) Parameters. The `dataAcquisitionModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Setting/Path	Type	Unit	Description
buffercount	int	-	The number of buffers used internally by the module for data recording.
buffersize	double	s	The buffersize of the module's internal data buffers.
triggered	bool	-	Indicates whether the module has recently triggered: 1=Yes, 0=No.

3.6. Device Settings Module

The Device Settings Module provides functionality for saving and loading device settings to and from file. The file is saved in [XML](#) format.

In general, users are recommended to use the utility functions provided by the APIs instead of using the Device Settings module directly. The Matlab API provides `ziSaveSettings()` and `ziLoadSettings()` and the Python API provides `zhinst.utils.save_settings()` and `zhinst.utils.load_settings`. These are convenient wrappers to the Device Settings module for loading settings synchronously, i.e., these functions block until loading or saving has completed, the desired behavior in most cases. Advanced users can use the Device Settings module directly if they need to implement loading or saving asynchronously (non-blocking).

See [Table 3.13](#) for the input parameters to configure the Device Settings Module.

Table 3.13. Device Settings Input Parameters

Setting/Path	Type	Description
<code>deviceSettings/device</code>	string	The device ID to save the settings for, e.g., <code>dev123</code> (compulsory parameter).
<code>deviceSettings/command</code>	string	The command to issue: "load" (load settings from file); "save" (read device settings and save to file) or "read" (just read the device settings) (compulsory parameter).
<code>deviceSettings/filename</code>	string	The name of the file to load or save to.
<code>deviceSettings/path</code>	string	The path containing the file to load from or save to.

Table 3.14. Device Settings Parameters reserved for use by the LabOne Web Server.

Setting/Path	Type	Description
<code>deviceSettings/throwonerror</code>	uint64	Throw an exception if there was error executing the command.
<code>deviceSettings/errortext</code>	string	The error text used in error messages.
<code>deviceSettings/finished</code>	uint64	The status of the command (read-only).

3.7. Impedance Module

The Impedance Module corresponds to the Cal sub-tab in the LabOne User Interface Impedance Analyzer tab. It allows the user to perform a compensation that will be applied to impedance measurements.

Table 3.15. Impedance Module Parameters.

Setting/Path	Type	Unit	Description
impedanceModule/ directory	string	-	The directory where files are saved.
impedanceModule/ calibrate	bool	-	If set to true will execute a compensation for the specified compensation condition.
impedanceModule/ device	string	-	Device string defining the device on which the compensation is performed.
impedanceModule/ step	int	-	Compensation step to be performed when calibrate indicator is set to true. step=0: First load; step=1: Second load; step=2: Third load; step=3: Fourth load.
impedanceModule/ mode	int	-	Compensation mode to be used. Defines which load steps need to be compensated: 3: SO (Short-Open) 4: L (Load) 5: SL (Short-Load) 6: OL (Open-Load) 7: SOL (Short-Open-Load) 8: LLL (Load-Load-Load)
impedanceModule/ status	int	-	Bit coded field of the already compensated load conditions (bit 0 = first load).
impedanceModule/ loads/0/r	double	Ohm	Resistance value of first compensation load (SHORT).
impedanceModule/ loads/1/r	double	Ohm	Resistance value of second compensation load (OPEN).
impedanceModule/ loads/2/r	double	Ohm	Resistance value of third compensation load (LOAD).
impedanceModule/ loads/3/r	double	Ohm	Resistance value of the fourth compensation load (LOAD). This load setting is only used if high impedance load is enabled.
impedanceModule/ loads/0/c	double	F	Parallel capacitance of the first compensation load (SHORT).
impedanceModule/ loads/1/c	double	F	Parallel capacitance of the second compensation load (OPEN).
impedanceModule/ loads/2/c	double	F	Parallel capacitance of the third compensation load (LOAD).
impedanceModule/ loads/3/c	double	F	Parallel capacitance of the fourth compensation load (LOAD).

Setting/Path	Type	Unit	Description
impedanceModule/ freq/start	double	Hz	Start frequency of compensation traces.
impedanceModule/ freq/stop	double	Hz	Stop frequency of compensation traces.
impedanceModule/ freq/samplecount	int	-	Number of samples of a compensation trace
impedanceModule/ highimpedanceload	bool	-	Enable a second high impedance load compensation for the low current ranges.
impedanceModule/ expectedstatus	int	-	Bit field of the load condition that the corresponds a full compensation. If status is equal the expected status the compensation is complete.
impedanceModule/ message	string	-	Message string containing information, warnings or error messages during compensation.
impedanceModule/ comment	string	-	Comment string that will be saved together with the compensation data.
impedanceModule/ validation	bool	-	Enable the validation of compensation data. If enabled the compensation is checked for too big deviation from specified load.
impedanceModule/ precision	int	-	Precision of the compensation. Will affect time of a compensation and reduces the noise on compensation traces, precision=0: Standard speed; precision=1: Low speed / high precision
impedanceModule/ todevice	bool	-	If enabled will automatically transfer compensation data to the persistent flash memory in case of a valid compensation.
impedanceModule/ progress	double	-	Progress of a compensation condition.

3.8. Multi-Device Synchronisation Module

The Multi-Device Synchronisation Module corresponds to the MDS tab in the LabOne User Interface. In essence, the module enables the clocks of multiple instruments to be synchronized such that timestamps of the same value delivered by different instruments correspond to the same point in time, thus allowing several instruments to operate in unison and their measurement results to be directly compared. The User Manual gives a more comprehensive description of multi-instrument synchronization, and also details the cabling required to achieve this.

Table 3.16. Multi-Device Synchronisation Module Parameters.

Setting/Path	Type	Unit	Description
<code>multiDeviceSyncModule/ devices</code>	string	-	Defines which instruments should be included in the synchronization. Expects a comma-separated list of devices in the order the devices are connected.
<code>multiDeviceSyncModule/ group</code>	int	-	Defines in which synchronization group should be accessed by the module.
<code>multiDeviceSyncModule/ message</code>	string	-	Status message of the module.
<code>multiDeviceSyncModule/ start</code>	bool	-	Set to true to start the synchronization process.
<code>multiDeviceSyncModule/ status</code>	int	-	Status of the synchronization process: -1 = error; 0 = idle; 1 = synchronization in progress; 2 = successful synchronization.

3.9. PID Advisor Module

The PID Advisor Module provides the functionality available in the Advisor, Tuner and Display sub-tabs of the LabOne User Interface's PID / PLL tab. The PID Advisor is a mathematical model of the instrument's PID and can be used to calculate PID controller parameters for optimal feedback loop performance. The controller gains calculated by the module can be easily transferred to the device via the API and the results of the Advisor's modelling are available as Bode and step response plot data as shown in [Figure 3.10](#).

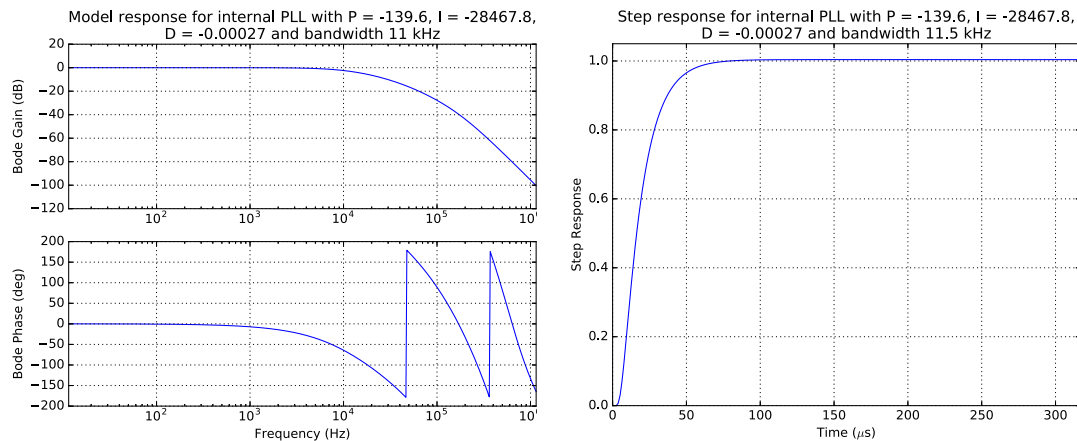


Figure 3.10. The plots generated by the LabOne Python API PID Advisor example (`example_pid_advisor_pll.py`) which configures the PID Advisor to optimize an internal PLL control loop. The data used in the Bode and step response plots is returned by the PID Advisor in the `pidAdvisor/bode` respectively `pidAdvisor/step` output parameters. See [Section 5.2.3](#) for help getting started with the Python examples.

3.9.1. PID Advisor Module Work-Flow

PID Advisor usage via the LabOne APIs closely follows the work-flow used in the LabOne User Interface. Here are the steps required to calculate optimal PID parameters, transfer the parameters to an instrument and then continually modify the instrument's parameters to minimize the residual error using Auto Tune.

1. Create an instance of the PID Advisor module.
2. Configure the module's parameters using `set()` to specify, for example, which of the instrument's PIDs to use and which type of device under test (DUT) to model. If values are specified for the P, I and D gains they serve as initial values for the optimization process. See [Table 3.21](#) for a full list of PID Advisor parameters.
3. Start the module by calling `execute()`.
4. Start optimization of the PID parameters by setting the `pidAdvisor/calculate` parameter to 1. The optimization process has finished when the value of `pidAdvisor/calculate` returns to 0. Optimization may take up to a minute to complete, but is much quicker in most cases.
5. Read out the optimized parameters, Bode and step response plot data (see [Figure 3.10](#)) for inspection using `get()`.
6. Transfer the optimized gain parameters from the Advisor Module to the instrument's nodes by setting the `pidAdvisor/todevice` parameter to 1.
7. Enable the instrument's PID (/DEV..../PIDS/n/ENABLE).
8. The Auto Tune functionality may be additionally enabled by setting `pidAdvisor/tune` to 1 and configuring the parameters in the `pidAdvisor/tuner` branch. This functionality

continuously updates the instrument's PID parameters specified by `pidAdvisor/tuner/mode` in order to minimize the residual error signal. Note, Auto Tune is not available for HF2 instruments.

The reader is encouraged to refer to the instrument-specific User Manual for more details on the Advisor optimization and Tuner process. Each of the LabOne APIs include an example to help get started programming with PID Advisor Module.

3.9.2. PLL Parameter Optimization on HF2 Instruments

On HF2 instruments the PID and PLL are implemented as two separate entities in the device's firmware. On all other devices there is only a PID unit and a PLL is created by configuring a PID appropriately (by setting the device node `/devN/pids/0/mode` to 1, see your instrument User Manual for more information). Since both a PID and a PLL exist on HF2 devices, when the PID Advisor Module is used to model a PLL, the `pidAdvisor/pid/type` parameter must be set to either `pid` or `pll` to indicate which hardware unit on the HF2 is to be modelled by the Advisor.

The Matlab and Python APIs have additional HF2-specific examples for using the PID Advisor Module with the HF2's PLL.

3.9.3. Instrument Settings written by `pidAdvisor/todevice`

This section lists which device nodes are configured upon setting the `pidAdvisor/todevice` parameter to 1. Note, the parameter is immediately set back to 0 and no `sync()` is performed, if a synchronization of instrument settings is required before proceeding, the user must execute a `sync` command manually.

For HF2 instruments there are two main cases to differentiate between, defined by whether `pidAdvisor/type` is set to "pid" or "pll" (see [Section 3.9.2](#) for an explanation). For UHF and MF devices `pidAdvisor/type` can only be set to "pid", for these devices [Table 3.17](#) and [Table 3.18](#) describe which device nodes are configured.

Table 3.17. The device nodes configured when `pidAdvisor/type` is "pid" (default behaviour). The value of `n` in device nodes corresponds to the value of `pidAdvisor/index`.

Device node (/DEV.../)	Value set (prefix <code>pidAdvisor/</code> omitted)	Device class
PIDS/n/P	Advised <code>pid/p</code> .	All devices
PIDS/n/I	Advised <code>pid/i</code> .	All devices
PIDS/n/D	Advised <code>pid/d</code> .	All devices
PIDS/n/DEMODO/ TIMECONSTANT	User-configured or advised <code>pid/ timeconstant</code> .	All devices
PIDS/n/DEMODO/ORDER	User-configured <code>pid/order</code> .	All devices
PIDS/n/DEMODO/HARMONIC	User-configured <code>pid/harmonic</code> .	All devices
PIDS/n/RATE	User-configured <code>pid/rate</code> .	Not HF2
PIDS/n/DLIMITTIMECONSTANT	User-configured or advised <code>pid/ dlimittimeconstant</code> .	Not HF2

Table 3.18. The additional device nodes configured when `pidAdvisor/type` is "pid" (default behaviour) and `pidAdvisor/dut/source=4` (internal PLL). The value of `n` in device nodes corresponds to the value of `pidAdvisor/index`.

Device node (/DEV.../)	Value set	Device class
PIDS/n/CENTER	User-configured <code>pidAdvisor/dut/ fcenter</code> .	All devices

Device node (/DEV.../)	Value set	Device class
PIDS/n/LIMITLOWER	Calculated $-\text{pidAdvisor}/\text{bw} \times 2$, if $\text{pidAdvisor}/\text{autolimit}=1$.	Not HF2
PIDS/n/LIMITUPPER	Calculated $\text{pidAdvisor}/\text{bw} \times 2$, if $\text{pidAdvisor}/\text{autolimit}=1$.	Not HF2
PIDS/n/RANGE	Calculated $\text{pidAdvisor}/\text{bw} \times 2$, if $\text{pidAdvisor}/\text{autolimit}=1$.	HF2 only

Table 3.19. The device nodes configured when `pidAdvisor/type` is "pll" (HF2 instruments only - see [Section 3.9.2](#) for an explanation). The value of `n` in device nodes corresponds to the value of `pidAdvisor/index`.

Device node (/DEV.../)	Value set
PLLS/n/AUTOTIMECONSTANT	Set to 0.
PLLS/n/AUTOPID	Set to 0.
PLLS/n/P	Advised <code>pidAdvisor/pid/p</code> .
PLLS/n/I	Advised <code>pidAdvisor/pid/i</code> .
PLLS/n/D	Advised <code>pidAdvisor/pid/d</code> .
PLLS/n/HARMONIC	Advised <code>pidAdvisor/demod/harmonic</code> .
PLLS/n/ORDER	Advised <code>pidAdvisor/demod/order</code> .
PLLS/n/TIMECONSTANT	User-configured or advised <code>pidAdvisor/demod/timeconstant</code> .

3.9.4. Monitoring the PID's Output

This section is not directly related to the functionality of the PID Advisor itself, but describes how to monitor the PID's behaviour by accessing the corresponding device's nodes on the Data Server.

MF and UHF Instruments

On MF and UHF instruments, the PID's error, shift and output value are available from the device's PID [streaming nodes](#):

- `/DEV.../PIDS/n/STREAM/ERROR`,
- `/DEV.../PIDS/n/STREAM/SHIFT`,
- `/DEV.../PIDS/n/STREAM/VALUE`.

These are high-speed streaming nodes with timestamps available for each value. They may be recorded using the [Data Acquisition Module](#) (recommended) or via the subscribe and poll commands (very high-performance applications). The PID streams are aligned by timestamp with demodulator streams. A specific streaming rate may be requested by setting the `/DEV.../PIDS/n/STREAM/RATE` node; the device firmware will set the next lowest configurable rate (which corresponds to a legal demodulator rate). The configured rate can be read out from the `/DEV.../PIDS/n/STREAM/EFFECTIVERATE` node. If the instrument has the DIG Option installed, the PID's outputs can also be obtained using the instrument's scope at rates of up to 1.8 GHz (although not continuously). Note, that `/DEV.../PIDS/n/STREAM/{RATE,EFFECTIVERATE}` do not effect the rate of the PID itself, only the rate at which data is transferred to the PC. The rate of an instrument's PID is configured by `/DEV.../PIDS/n/RATE`.

HF2 Instruments

On HF2 instruments the PID's error, shift and center values are available using the device nodes:

- /DEV..../PIDS/n/ERROR (output node),
- /DEV..../PIDS/n/SHIFT (output node),
- /DEV..../PIDS/n/CENTER (setting node),

where the PID's output can be calculated as $OUT = CENTER + SHIFT$. When data is acquired from these nodes using the subscribe and poll commands the node values do not have timestamps associated with them (since the HF2 Data Server only supports [API Level 1](#)). Additionally, these nodes are not high-speed [streaming nodes](#); they are updated at a low rate that depends on the rate of the PID, this is a approximately 10 Hz if one PID is enabled. It is not possible to configure the rate of the PID on HF2 instruments. It is possible, however, to subscribe to the /DEV.../PIDS/n/ERROR node in the [Data Acquisition Module](#), here, timestamps are approximated for each error value. It is not possible to view these values in the HF2 scope.

The PLL Module

The PLL Advisor Module introduced in LabOne 14.08 became deprecated as of LabOne 16.12. In LabOne 16.12 the PLL Advisor's functionality was combined within the PID Advisor module. Users of the PLL Advisor Module should use the PID Advisor Module instead.

3.9.5. PID Advisor Module Parameters

The following tables provide a comprehensive list of the module's parameters, see:

- [Table 3.20](#) for input/output parameters,
- [Table 3.21](#) for input parameters and,
- [Table 3.22](#) for output parameters.

Table 3.20. PID Advisor Input/Output Parameters. The `pidAdvisor/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Path	Type	Unit	Description
calculate	int	-	Set to 1 to start the PID Advisor's modelling process and calculation of optimal parameters. The module sets <code>calculate</code> to 0 when the calculation is finished.
response	int	-	Set to 1 to calculate the Bode and the step response plot data from the current <code>pid/*</code> parameters (only relevant when <code>auto=0</code>). The module sets <code>response</code> back to 0 when the plot data has been calculated.
todevice	int	-	Set to 1 to transfer the calculated PID advisor data to the device, the module will immediately reset the parameter to 0 and configure the instrument's nodes, see Section 3.9.3 for more information.
pid/d	double	(Output Unit . s)/ Input Unit	The initial value to use in the Advisor for the differential gain. After optimization has finished it contains the optimal value calculated by the Advisor.
pid/i	double	Output Unit/(Input Unit . s)	The initial value to use in the Advisor for the integral gain. After optimization has finished

Path	Type	Unit	Description
			it contains the optimal value calculated by the Advisor.
pid/p	double	Output Unit/ Input Unit	The initial value to use in the Advisor for the proportional gain. After optimization has finished it contains the optimal value calculated by the Advisor.
pid/dlimittimeconstant	double	s	The initial value to use in the Advisor for the differential filter timeconstant gain. After optimization has finished it contains the optimal value calculated by the Advisor.

Table 3.21. PID Advisor Input Parameters. The `pidAdvisor/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Path	Type	Unit	Description
advancedmode	int	-	If enabled, automatically calculate the start and stop value used in the Bode and step response plots.
auto	int	-	If enabled, automatically trigger a new optimization process upon an input parameter value change.
demod/harmonic	int	-	Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output. Specifies the demodulator's harmonic to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV.../DEMOS/m/HARMONIC) when the PID is enabled.
demod/order	int	-	Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output. Specifies the demodulator's order to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV.../DEMOS/m/ORDER) when the PID is enabled.
demod/timeconstant	double	s	Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output and <code>pidAdvisor/pid/autobw=0</code> . Specify the demodulator's timeconstant to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV.../DEMOS/m/TIMECONSTANT) when the PID is enabled.
display/freqstart	double	Hz	Start frequency for Bode plot. If <code>advancedmode=0</code> the start value is automatically derived from the system properties.
display/freqstop	double	Hz	Stop frequency for Bode plot.
display/timestart	double	s	Start time for step response. If <code>advancedmode=0</code> the start value is 0.
display/timestop	double	s	Stop time for step response.

Path	Type	Unit	Description
dut/bw	double	Hz	Bandwidth of the DUT (device under test).
dut/damping	double	-	Damping of the second order low pass filter.
dut/delay	double	s	IO Delay of the feedback system describing the earliest response for a step change.
dut/fcenter	double	Hz	Resonant frequency of the of the modelled resonator.
dut/gain	double	Depends on Input, Output and DUT model	Gain of the DUT transfer function.
dut/q	double	-	Quality factor of the modelled resonator.
dut/source	int	-	Specifies the model used for the external DUT (device under test) to be controlled by the PID: 1: Low-pass first order. 2: Low-pass second order. 3: Resonator frequency. 4: Internal PLL. 5: Voltage-controlled oscillator (VCO). 6: Resonator amplitude.
index	int	-	The 0-based index of the PID on the instrument to use for parameter detection.
pid/autobw	int	-	If enabled, adjust the demodulator bandwidth to fit best to the specified target bandwidth of the full system. In this case, <code>pidAdvisor/demod/timeconstant</code> is ignored.
pid/autolimit	int	-	If enabled, set the instrument PID limits based upon the calculated bw value. See Table 3.18 for more information.
pid/mode	double	-	Select PID Advisor mode; bit encoded: bit 0: Optimize P gain. bit 1: Optimize I gain. bit 2: Optimize D gain. bit 3: Optimize D filter limit.
pid/rate	double	Hz	PID Advisor sampling rate of the PID control loop.
pid/targetbw	double	Hz	PID system target bandwidth.
pid/type	string	-	HF2 instruments only. Specify whether to model the instrument's PLL or PID hardware unit when <code>pidAdvisor/dut/source=4</code> (internal PLL). See Section 3.9.2 .
targetbw	double	Hz	Requested PID bandwidth. Higher frequencies may need manual tuning.

Path	Type	Unit	Description
tf/closedloop	int	-	Switch the response calculation mode between closed or open loop.
tf/input	int	-	Start point for the plant response simulation for open or closed loops.
tf/output	int	-	End point for the plant response simulation for open or closed loops.
tune	int	-	If enabled, optimize the instrument's PID parameters so that the noise of the closed-loop system gets minimized. The HF2 doesn't support tuning.
tuner/mode	int	-	Select tuner mode; bit encoded: bit 0: Tune P gain. bit 1: Tune I gain. bit 2: Tune D gain. bit 3: Tune D filter limit.
tuner/averagetime	double	s	Time for a tuner iteration.

Table 3.22. PID Advisor Output (read-only) Parameters. The `pidAdvisor/` prefix is omitted from the parameter paths.

Path	Type	Unit	Description
bode	struct	-	The resulting Bode plot data of the PID Advisor's simulation, see Figure 3.10 for an example. Contains the following fields: flags Reserved for future use. grid An array containing the frequency axis values for the complex Bode data. samplecount Reserved for future use. sampleformat Indicates that this struct contains Bode plot data (always equal to 0). x An array containing the real component values of the complex Bode data. y An array containing the imaginary component values of the complex Bode data.
bw	double	Hz	Calculated system bandwidth.
impulse	struct	-	Reserved for future use - not yet supported.
pm	double	deg	Simulated phase margin of the PID with the current settings. The phase margin should be greater than 45 deg and preferably greater than 65 deg for stable conditions.
pmfreq	double	Hz	Simulated phase margin frequency.

Path	Type	Unit	Description
stable	int	-	If equal to 1, the PID Advisor found a stable solution with the given settings. If equal to 0, the solution was deemed instable - revise your settings and rerun the PID Advisor.
step	struct	-	The resulting step response data of the PID Advisor's simulation, see Figure 3.10 for an example. Contains the following fields: flags Reserved for future use. grid An array containing the time axis values for the step response data. samplecount Reserved for future use. sampleformat Indicates that this struct contains step response plot data (always equal to 1). x An array containing the values of the step response. y Unused (all values=0).
targetfail	int	-	A value of 1 indicates the simulated PID BW is smaller than the Target BW.

3.10. Scope Module

The Scope Module corresponds to the functionality available in the Scope tab in the LabOne User Interface and provides API users with an interface to acquire assembled and scaled scope data from the instrument programmatically.

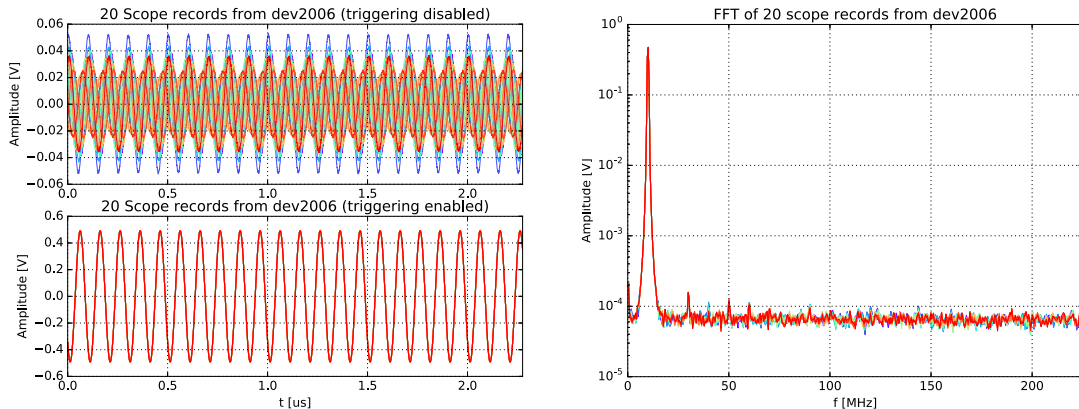


Figure 3.11. The plots generated by the LabOne Python API Scope Module example when ran with a UHF Instrument (`example_scope.py`). The example runs the Scope Module in both time and frequency mode with scope record averaging enabled. See [Section 5.2.3](#) for help getting started with the Python examples.

3.10.1. Introduction to Scope Data Transfer

In general, an instrument's scope can generate a large amount of data which requires special treatment by the instrument's firmware, the Data Server, LabOne API and API client in order to process it correctly and efficiently. The Scope Module was introduced in LabOne 16.12 to simplify scope data acquisition for the user. This section provides a top-level overview of how scope data can be acquired and define the terminology used in subsequent sections before looking at the special and more simplified case when the Scope Module is used.

There are three methods of obtaining scope data from the device:

1. By subscribing directly to the instrument node `/DEV.../SCOPES/n/WAVE` and using the `poll()` command. This refers to the lower-level interface provided by the `ziDAQServer` class `subscribe()` and `poll()` commands.
2. By subscribing to the instrument node `/DEV.../SCOPES/n/WAVE` in the Scope Module and using the Scope Module's `read()` command.
3. By subscribing to the instrument's scope streaming node (`/DEV.../SCOPES/n/STREAM/SAMPLE`), which continuously streams scope data as a block stream. This is only available on MF and UHF instruments with the DIG Option enabled. The Scope Module does not support acquisition from the scope streaming node.

Segmented Mode

Additionally, MF and UHF instruments which have the DIG option enabled can optionally record data in "segmented" mode which allows back-to-back measurements with very small hold-off times between measurements. Segmented mode enables the user to make a fixed number of

measurements (the segments), which are stored completely in the memory of the instrument, before they are transferred to the Data Server (this overcomes the data transfer rate limitation of the device's connected physical interface, e.g., USB or 1GbE for a fixed number of measurements). The advantage to this mode is precisely that the hold-off time, i.e. the delay between two measurements, can be very low. Data recorded in segmented mode is still available from the instrument node `/DEV..../SCOPES/n/WAVE` as for non-segmented data, but requires an additional reshaping described in [Section 3.10.6](#).

Scope Data Nomenclature

We'll use the following terminology to describe the scope data streamed from the device:

Wave	The name of the leaf (<code>/DEV..../SCOPES/n/WAVE</code>) in the device node tree that contains scope data pushed from the instrument's firmware to the Data Server. The data structure returned by this node is defined by the API Level of the connected session. It is also the name of the structure member in scope data structures that holds the actual scope data samples. See Scope Data Structures below for detailed information.
Record	Refers to one complete scope data element returned by the Scope Module. It may consist of one or multiple segments.
Segment	A segment is a completely assembled and scaled wave. If the instrument's scope is used in segmented mode, each record will consist of multiple segments. If not used in segmented mode, each record comprises of a single segment and the terms record and segment can be used interchangeably.
Block	When the length of data (<code>/DEV..../SCOPES/n/LENGTH</code>) in a scope segment is very large the segment returned by the device node (<code>/DEV..../SCOPES/n/WAVE</code>) is split into multiple blocks. When using the poll/subscribe method the user must assemble these blocks; the Scope Module assembles them for the user into complete segments.
Shot	The term shot is often used when discussing data acquired from laboratory oscilloscopes, we try to avoid it in the following in order to more easily distinguish between records and segments when recording in segmented mode.

Scope Data Structures

The device node `/DEV..../SCOPES/n/WAVE` (and `/DEV..../SCOPES/n/STREAM/SAMPLE`, DIG Option enabled, API level > 4) return the following data structures based on the API level used by the session:

ScopeWave	API Level 1, HF2 only. The simplest scope data structure. The wave structure member in the data structure always has a fixed length of 2048. Scope records are never split into multiple blocks; no assembly required. The HF2 does not support segmented recording; a segment is equivalent to a record.
ZIScopeWave	API Level 4. An extended scope data structure used with MF and UHF instruments. The data in the wave structure member consists of one scope block; for long scope segments, complete scope segments must be assembled by combining these blocks. The data in wave is not scaled or offset.
ZIScopeWaveEx	API Level > 5. As for ZIScopeWave , but contains the additional structure member <code>channelOffset</code> .

The Scope Module always returns scope data in the [ZIScopeWaveEx](#) format, regardless of which supported API level (1, >4) was used in the session where the Scope Module was instantiated. However, the data in the wave structure member always consists of complete segments (it does not need to be reassembled from multiple blocks). More differences between the data returned

by the node `/DEV..../SCOPES/n/WAVE` and the Scope Module are highlighted in [3.10.2 Advantages of the Scope Module](#).

3.10.2. Advantages of the Scope Module

Although it is possible to acquire scope data using the lower-level subscribe/poll method, the Scope Module provides API users with several advantages. Specifically, the Scope Module:

1. Provides a uniform interface to acquire scope data from all instrument classes (HF2 scope usage differs from MF and UHF devices, especially with regards to scaling).
2. Scales and offsets the scope wave data to get physically meaningful values. If data is polled from the device node using subscribe/poll the scaling and offset must be applied manually.
3. Assembles large multi-block transferred scope data into single complete records. When the scope is configured to record large scope lengths and data is directly polled from the device node `/DEV.../SCOPES/n/WAVE`, the data is split into multiple blocks for efficient transfer of data from the Data Server to the API; these must then be programatically reassembled. The Scope Module performs this assembly and returns complete scope records (unless used in pass-through mode, `scopeModule/mode=0`).
4. Can be configured to return the FFT of the acquired scope records (with `scopeModule/mode=3`) as provided by the Scope Tab in the LabOne UI. FFT data is not available from the device nodes in the `/DEV..../SCOPES/` branch using subscribe/poll.
5. Can be configured to average the acquired scope records the `scopeModule/averager/` parameters.
6. Can be configured to return a specific number of scope records using the `scopeModule/historylength` parameter.

3.10.3. Working with Scope Module

It is important to note that the instrument's scope is implemented in the firmware of the instrument itself and most of the parameters relevant to scope data recording are configured as device nodes under the `/DEV.../SCOPES/` branch. Please refer to the instrument-specific User Manual for a description of the Scope functionality and a list of the available nodes.

The Scope Module does not modify the instrument's scope configuration and, as such, processes the data arriving from the instrument in a somewhat passive manner. The Scope Module simply reassembles data transferred in multiple blocks into complete segments (so that they consist of the configured `/DEV..../SCOPES/n/LENGTH`) and applies the offset and scaling required to get physically meaningful values to the (integer) data sent by the instrument.

The following steps should be used as a guideline for a Scope Module work-flow:

1. Create an instance of the Scope Module. This instance may be re-used for recording data with different instrument settings or Scope Module configurations.
2. Subscribe in the Scope Module to the scope's streaming block node (`/DEV..../SCOPES/n/WAVE`) to specify which device and scope to acquire data from. Data will only be acquired after enabling the scope and calling `Scope Module execute()`.
3. Configure the instrument ready for the experiment. When acquiring data from the signal inputs it is important to specify an appropriate value for the input range (`/DEV..../SIGINS/n/RANGE`) to obtain the best bit resolution in the scope. The signal input range on MF and UHF instruments can be adjusted automatically, see the `/DEV..../SIGINS/n/AUTORANGE` node and API utility functions demonstrating its use (e.g. `zhinst.utils.sign_autorange()` in the LabOne Python API).
4. Configure the instrument's scope as required for the measurement. If recording signals other than hardware channel signals (such as a PID's error or a demodulator R output), be sure to

- configure the `/DEV.../SCOPES/n/CHANNELS/n/LIMIT{LOWER,UPPER}` accordingly to obtain the best bit resolution in the scope).
5. Configure the `scopeModule/*` parameters as required, in particular:
 - Set `scopeModule/mode` in order to specify whether to return time or frequency domain data. See [Section 3.10.4](#) for more information on the Scope Module's modes.
 - Set the `scopeModule/historylength` parameter to tell the Scope Module to only return a certain number of records. Note, as the Scope Module is acquiring data the `scopeModule/records` output parameter may grow larger than `scopeModule/historylength`; the Scope Module will return the last number of records acquired.
 - Set `scopeModule/averager/weight` to a value larger than 1 to enable averaging of scope records, see [Section 3.10.5](#).
 6. Enable the scope (if not previously enabled) and call Scope Module `execute()` to start acquiring data.
 7. Wait for the Scope Module to acquire the specified number of records. Note, if certain scope parameters are modified during recording, the history of the Scope Module will be cleared, see [Section 3.10.7](#) for the list of parameters that trigger a Scope Module reset.
 8. Call Scope Module `read()` to transfer data from the Module to the client. Data may be read out using `read()` before acquisition is complete (advanced use). Note, an intermediate read will create a copy in the client of the incomplete record which could be critical for memory consumption in the case of very long scope lengths or high segment counts. The data structure returned by `read()` is of type `ZIScopeWaveEx`, see [Section 8.3.29](#) for detailed information.
 9. Check the flags of each record indicating whether any problems occurred during acquisition.
 10. Extract the data for each recorded scope channel and, if recording data in segmented mode, reshape the wave data to allow easier access to multi-segment records (see [Section 3.10.6](#)). Note, the scope data structure only returns data for enabled scope channels.

Data Acquisition and Transfer Speed

It is important to note that the time to transfer long scope segments from the device to the Data Server can be much longer than the duration of the scope record itself. This is not due to the Scope Module but rather due to the limitation of the physical interface that the device is connected on (USB, 1GbE). Please ensure that the PC being used has adequate memory to hold the scope records.

3.10.4. Scope Module Modes

The `scopeModule/mode` is applied for all scope channels returned by the Scope Module. Although it is not possible to return both time and frequency domain data in one instance of the Scope Module, multiple instances may be used to obtain both.

The Scope Module does not return an array consisting of the points in time (time mode) or frequencies (FFT mode) corresponding to the samples in `ZIScopeWaveEx`. These can be constructed as arrays of n points, where n is the configured scope length (`/DEV.../SCOPES/n/LENGTH`), spanning the intervals:

Time mode `[0, dt*totalsamples]`, where `dt` and `totalsamples` are fields in `ZIScopeWaveEx`.

In order to get a time array relative to the trigger position, `(timestamp - triggertimestamp)/float(clockbase)` must be subtracted from the times in the interval, where `timestamp` and `triggerstamp` are fields in `ZIScopeWaveEx`.

FFT mode $[0, (\text{clockbase}/2^{\text{scope\_time}})/2]$, where `scope_time` is the value of the device node `/DEV..../SCOPES/n/TIME`.

and `clockbase` is the value of `/DEV..../CLOCKBASE`.

3.10.5. Averaging

When `scopeModule/averager/weight` is set to be greater than 1, then each new scope record in the history element is an exponential moving average of all the preceding records since either `execute()` was called or `scopeModule/averager/restart` was set to 1. The average is calculated by the Scope Module as following:

```
alpha = 2/(weight + 1);
newVal = alpha * lastRecord + (1 - alpha) * history;
history = newVal
```

where `newVal` becomes the last record in the Scope Module's history. If the scope is in Single mode, no averaging is performed. The weight corresponds to the number of segments to achieve 63% settling, doubling the value of weight achieves 86% settling.

It is important to note that the averaging functionality is performed by the Scope Module on the PC where the API client is ran, not on the device. Enabling averaging does not mean that less data is sent from the instrument to the Data Server.

The average calculation can be restarted by setting `scopeModule/averager/restart` to 1. It is currently not possible to tell how many scope segments have been averaged (since the reset). In general, however, the way to track the current scope record is via the `sequenceNumber` field in [ZIScopeWaveEx](#).

3.10.6. Segmented Recording

When the instrument's scope is ran in segmented mode the `wave` structure member in the [ZIScopeWaveEx](#) is an array consisting of `length*segment_count` where `length` is the value of `/DEV..../SCOPES/0/LENGTH` and `num_segments` is `/DEV..../SCOPES/n/SEGMENTS/COUNT`. This is equal to the value of the `totalSamples` structure member.

The Scope Module's `progress()` method can be used to check the progress of the acquisition of a single segmented recording. It is possible to read out intermediate data before the segmented recording has finished. This will however, perform a copy of the data; the user should ensure he has adequate memory available.

If the segment count in the instrument's scope is changed, the Scope Module must be re-executed. It is not possible to read multiple records consisting of different numbers of segments within one Scope Module execution.

3.10.7. Scope Parameters that reset the Scope Module

The Scope Module parameter `scopeModule/records` and `progress` are reset and all records are cleared from the Scope Module's history when the following critical instrument scope settings are changed:

- `/DEV..../SCOPES/n/LENGTH`
- `/DEV..../SCOPES/n/RATE`
- `/DEV..../SCOPES/n/CHANNEL`
- `/DEV..../SCOPES/n/SEGMENTS/COUNT`
- `/DEV..../SCOPES/n/SEGMENTS/ENABLE`

3.10.8. Device-specific considerations

Scope Module Use on HF2 Instruments

The HF2 scope is supported by the Scope Module, in which case the API connects to the HF2 Data Server using [API Level 1](#) (the HF2 Data Server does not support higher levels). When using the Scope Module with HF2 Instruments the parameter `scopeModule/externalscaling` must be additionally configured based on the currently configured scope signal's input/output hardware range, see the `externalscaling` entry in [Table 3.24](#) for more details. This is not necessary for other instrument classes.

Scope Module Use on MF and UHF Instruments

For MF and UHF instruments no special considerations must be made except that [API Level 4](#) is not supported by the Scope Module; a higher API level must be used.

Whilst not specific to the Scope Module, it should be noted that the instrument's scope is not effected by loading device presets, in particular, default scope settings are not obtained by loading the instrument's factory preset.

3.10.9. Scope Module Parameters

The following tables provide a comprehensive list of the module's parameters, see:

- [Table 3.23](#) for input/output parameters,
- [Table 3.24](#) for input parameters and,
- [Table 3.25](#) for output parameters.

Table 3.23. Scope Module Input/Output Parameters. The `scopeModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Path	Type	Unit	Description
<code>averager/restart</code>	bool	-	Set to 1 to reset the averager. The module sets <code>averager/restart</code> back to 0 automatically.

Table 3.24. Scope Module Parameters. The `scopeModule/` prefix is omitted from the parameter paths, this must be included in the path when setting parameters.

Path	Type	Unit	Description
<code>averager/weight</code>	int	-	Specify the averaging behaviour: <code>weight=1</code> Averaging disabled. <code>weight>1</code> Exponentially average the incoming scope records, updating the last scope record in the history with the averaged record, see Section 3.10.5.
<code>averager/resamplingmode</code>	int	-	Specifies the resampling mode. When averaging scope data recorded at a low sampling rate that is aligned by a high

Path	Type	Unit	Description
			resolution trigger, scope data must be resampled to keep the corresponding samples between averaged recordings aligned correctly in time relative to the trigger time. The mode is either: 0: Linear interpolation, 1: PCHIP interpolation.
<code>clearhistory</code>	bool	-	Remove all records from the history list.
<code>externalscaling</code>	double	-	Only relevant for HF2 Instruments. Specify the scaling to apply to the scope data based on the range of the instrument channel (/DEV.../SCOPES/0/CHANNEL) that is being recorded. For the following values of /DEV.../SCOPES/0/CHANNEL set <code>externalscaling</code> to the value of the corresponding device RANGE node: 0: /DEV.../SIGINS/0/RANGE 1: /DEV.../SIGINS/1/RANGE 2: /DEV.../SIGOUTS/0/RANGE 3: /DEV.../SIGOUTS/1/RANGE
<code>fft/power</code>	bool	-	Enable calculation of the power value.
<code>fft/spectraldensity</code>	bool	-	Enable calculation of the spectral density value.
<code>fft/window</code>	int	-	FFT Window: window=0: Rectangular; window=1: Hann (default); window=2: Hamming; windows=3: Blackman Harris.
<code>historylength</code>	int	-	Maximum number of entries stored in the measurement history.
<code>lastreplace</code>	int	-	Reserved for LabOne User Interface use.
<code>mode</code>	int	-	The Scope Module's data processing mode: 0: Pass-through: scope segments assembled and returned unprocessed, non-interleaved. 1: Moving average: entire scope recording assembled, scaling applied, averager if enabled (see <code>averager/weight</code>), data returned in float non-interleaved format. 2: Reserved for future use (average <i>n</i> segments). 3: FFT, same as mode 1, except an FFT is applied to every segment of the scope recording before averaging. See the <code>fft/*</code> parameters for FFT parameters.

Path	Type	Unit	Description
save/*	-	-	Core module save/ branch parameters, see Table 3.1 and Table 3.2 for a description of the parameters.

Table 3.25. Scope Module Output (read-only) Parameters. The `scopeModule/` prefix is omitted from the parameter paths.

Path	Type	Unit	Description
error	int	-	Indicates whether an error occurred whilst processing the current scope record; set to non-zero when a scope flag indicates an error. The value indicates the accumulated error for all the processed segments in the current record and is reset for every new incoming scope record. It corresponds to the status LED in the LabOne User Interface's Scope tab - API users are recommended to use the <code>flags</code> structure member in ZIScopeWaveEx instead of this output parameter.
records	int	-	The number of scope records that have been processed by the Scope Module since <code>execute()</code> was called or a critical scope setting has been modified (see Section 3.10.7 for a list of scope settings that trigger a reset).

3.11. Sweeper Module

The Sweeper Module allows the user to perform sweeps as in the Sweeper Tab of the LabOne User Interface. In general, the Sweeper can be used to obtain data when measuring a DUT's response to varying (or **sweeping**) one instrument setting while other instrument settings are kept constant.

3.11.1. Configuring the Sweeper

In the following we briefly describe how to configure the Sweeper Module. See [Table 3.26](#) for a full list of the Sweeper's parameters and [Table 3.27](#) for a description of the Sweeper's outputs.

Specifying the Instrument Setting to Sweep

The Sweeper's `sweep/gridnode` parameter, the so-called **sweep parameter**, specifies the instrument's setting to be swept, specified as a path to an instrument's [node](#). This is typically an oscillator frequency in a [Frequency Response Analyzer](#), e.g., `/dev123/oscs/0/freq`, but a wide range of instrument settings can be chosen, such as a signal output amplitude or a PID controller's setpoint.

Specifying the Range of Values for the Sweep Parameter

The Sweeper will change the sweep parameter's value `sweep/samplecount` times within the **range** of values specified by `sweep/start` and `sweep/stop`. The `sweep/xmapping` parameter specifies whether the spacing between two sequential values in the range is linear (`=0`) or logarithmic (`=1`).

Controlling the Scan mode: The Selection of Range Values

The `sweep/scan` parameter defines the **order** that the values in the specified range are written to the sweep parameter. In sequential scan mode (`=0`), the sweep parameter's values change incrementally from smaller to larger values, see [Figure 3.14](#). In order to scan the sweep parameter's in the opposite direction, i.e., from larger to smaller values, reverse scan mode (`=3`) can be used.

In binary scan mode (`=1`) the first sweep parameter's value is taken as the value in the middle of the range, then the range is split into two halves and the next two values for the sweeper parameter are the values in the middle of those halves. This process continues until all the values in the range were assigned to the sweeper parameter, see [Figure 3.16](#). Binary scan mode ensures that the sweep parameter uses values from the entire range near the beginning of a measurement, which allows the user to get feedback quickly about the measurement's entire range. Since the Sweeper Module is an [asynchronous](#) interface, it's possible to continuously read and plot data whilst the sweep measurement is ongoing and update points in a graph dynamically.

In bidirectional scan mode (`=2`) the sweeper parameter's values are first set from smaller to larger values as in sequential mode, but are then set in reverse order from larger to smaller values, see [Figure 3.15](#). This allows for effects in the sweep parameter to be observed that depend on the order of changes in the sweep parameter's values.

Controlling how the Sweeper sets the Demodulator's Time Constant

The `sweep/bandwidthcontrol` parameter specifies which demodulator filter bandwidth (equivalently time constant) the Sweeper should set for the current measurement point. The user can either specify the bandwidth manually (`=0`), in which case the value of the current demodulator filter's bandwidth is simply used for all measurement points; specify a fixed

bandwidth (=1), specified by `sweep/bandwidth`, for all measurement points; or specify that the Sweeper sets the demodulator's bandwidth automatically (=2). Note, to use either Fixed or Manual mode, `sweep/bandwidth` must be set to a value > 0 (even though in manual mode it is ignored).

Specifying the Sweeper's Settling Time

For each change in the sweep parameter that takes effect on the instrument the Sweeper waits before recording measurement data in order to allow the measured signal to settle. This behavior is configured by two parameters in the `sweep/settling/` branch: `sweep/settling/time` and `sweep/settling/inaccuracy`.

The `sweep/settling/time` parameter specifies the minimum time in seconds to wait before recording measurement data for that sweep point. This can be used to specify the settling time required by the user's experimental setup before measuring the response in their system.

The `sweep/settling/inaccuracy` parameter is used to derive the settling time to allow for the lock-in amplifier's demodulator filter response to settle following a change of value in the sweep parameter. More precisely, the `sweep/settling/inaccuracy` parameter specifies the amount of settling time as the time required to attain the specified remaining proportion [1e-13, 0.1] of an incoming step function. Based upon the value of `sweep/settling/inaccuracy` and the demodulator filter order, the number of demodulator filter time constants to wait is calculated and written to `sweep/settling/tc` (upon calling the module's `execute()` command) which can then be read back by the user. See [Table 3.26](#) for recommended values of `sweep/settling/inaccuracy`. The relationship between `sweep/settling/inaccuracy` and `sweep/settling/tc` is plotted in [Figure 3.12](#).

The actual amount of time the Sweeper Module will wait after setting a new sweep parameter value before recording measurement data is defined in [Equation 3.1](#). For a frequency sweep, the `sweep/settling/inaccuracy` parameter will tend to influence the settling time at lower frequencies, whereas `sweep/settling/time` will tend to influence the settling time at higher frequencies.

$$t_s = \max(\text{sweep\_settling\_tc} \times t_c, \text{sweep\_settling\_time})$$

Equation 3.1. The settling time t_s used by the Sweeper for each measurement point; the amount of time between setting the sweep parameter and recording measurement data is determined by the `sweep/settling/tc` and `sweep/settling/time`.

Note, although it is recommended to use `sweep/settling/inaccuracy`, it is still possible to set the settling time via `sweep/settling/tc` instead of `sweep/settling/inaccuracy` (the parameter applied will be simply the last one that is set by the user).

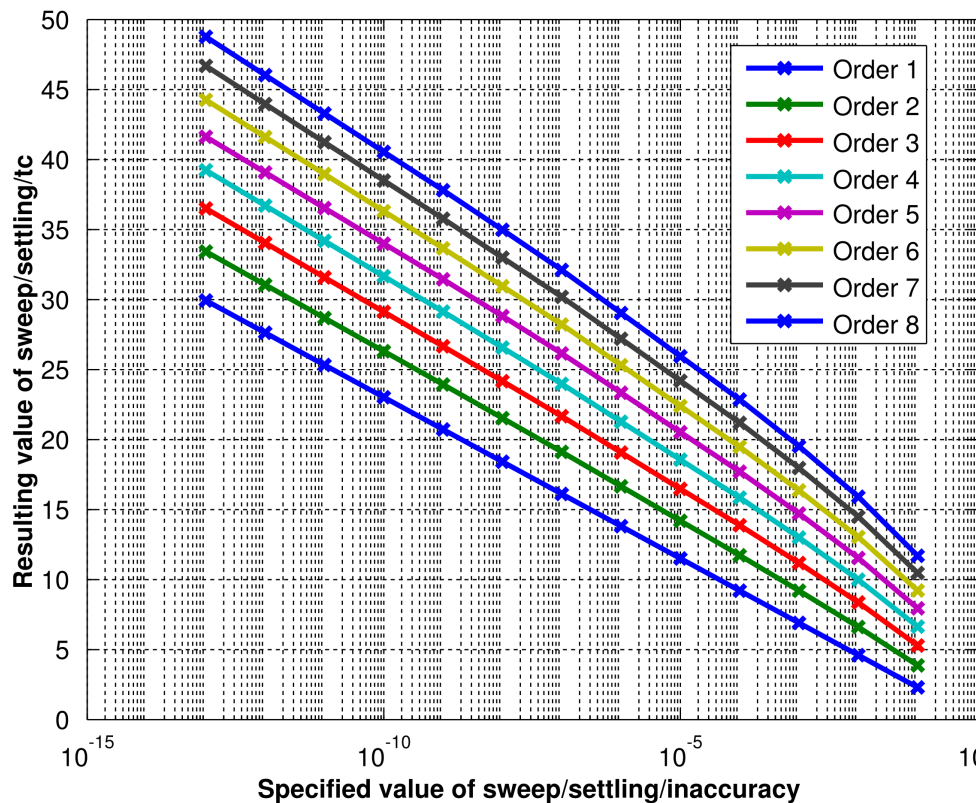


Figure 3.12. A plot showing the values of the Sweeper's sweep/settling/tc as calculated from sweep/settling/inaccuracy parameter and their dependency on demodulator order filter.

Specifying which Data to Measure

Which measurement data is actually returned by the Sweeper's read command is configured by subscribing to `node paths` using the Sweeper Module's `subscribe` command.

Specifying how the Measurement Data is Averaged

One Sweeper measurement point is obtained by averaging recorded data which is configured via the parameters in the `sweep/averaging/` branch.

The `sweep/averaging/tc` parameter specifies the minimum time window in factors of demodulator filter time constants during which samples will be recorded in order to average for one returned sweeper measurement point. The `sweep/averaging/sample` parameter specifies the minimum number of data samples that should be recorded and used for the average. The Sweeper takes both these settings into account for the measurement point's average according to Equation 3.2.

$$N = \max(\text{averaging\_tc} \times tc \times \text{sampling\_rate}, \text{averaging\_sample})$$

Equation 3.2. The number of samples N used to average one sweeper measurement point is determined by the parameters `sweep/averaging/tc` and `sweep/averaging/sample`.

Note, the value of the demodulator filter's time constant may be controlled by the Sweeper depending on the value of `sweep/bandwidthcontrol` and `sweep/bandwidth`, see [above](#), [Controlling how the Sweeper sets the Demodulator's Time Constant](#). For a frequency sweep, the

`sweep/averaging/tc` parameter will tend to influence the number of samples recorded at lower frequencies, whereas `sweep/averaging/sample` will influence averaging behavior at higher frequencies.

An Explanation of Settling and Averaging Times in a Frequency Sweep

Figure 3.13 shows which demodulator samples are used in order to calculate an averaged measurement point in a frequency sweep. This explanation of the Sweeper's parameters is specific to the following commonly-used Sweeper settings:

- `sweep/gridnode` is set to an oscillator frequency, e.g., `/dev123/oscs/0/freq`.
- `sweep/bandwidthcontrol` is set to 2, corresponding to automatic bandwidth control, i.e., the Sweeper will set the demodulator's filter bandwidth settings optimally for each frequency used.
- `sweep/scan` is set to 0, corresponding to sequential scan mode for the range of frequency values swept, i.e, the frequency is increasing for each measurement point made.

Each one of the three red segments in the demodulator data correspond to the data used to calculate one single Sweeper measurement point. The light blue bars correspond to the time the sweeper should wait as indicated by `sweep/settling/tc` (this is calculated by the Sweeper Module from the specified `sweep/settling/inaccuracy` parameter). The purple bars correspond to the time specified by the `sweep/settling/time` parameter. The sweeper will wait for the maximum of these two times according to Equation 3.1. When measuring at lower frequencies the Sweeper sets a smaller demodulator filter bandwidth (due to automatic `sweep/bandwidthcontrol`) corresponding to a larger demodulator filter time constant. Therefore, the `sweep/settling/tc` parameter dominates the settling time used by the Sweeper at low frequencies and at high frequencies the `sweep/settling/time` parameter takes effect. Note, that the light blue bars corresponding to the value of `sweep/settling/tc` get shorter for each measurement point (larger frequency used $\rightarrow$ shorter time constant required), whereas the purple bars corresponding to `sweep/settling/time` stay a constant length for each measurement point. Similarly, the `sweep/averaging/tc` parameter (yellow bars) dominates the Sweeper's averaging behavior at low frequencies, whereas `sweep/averaging/samples` (green bars) specifies the behavior at higher frequencies, see also Equation 3.2.

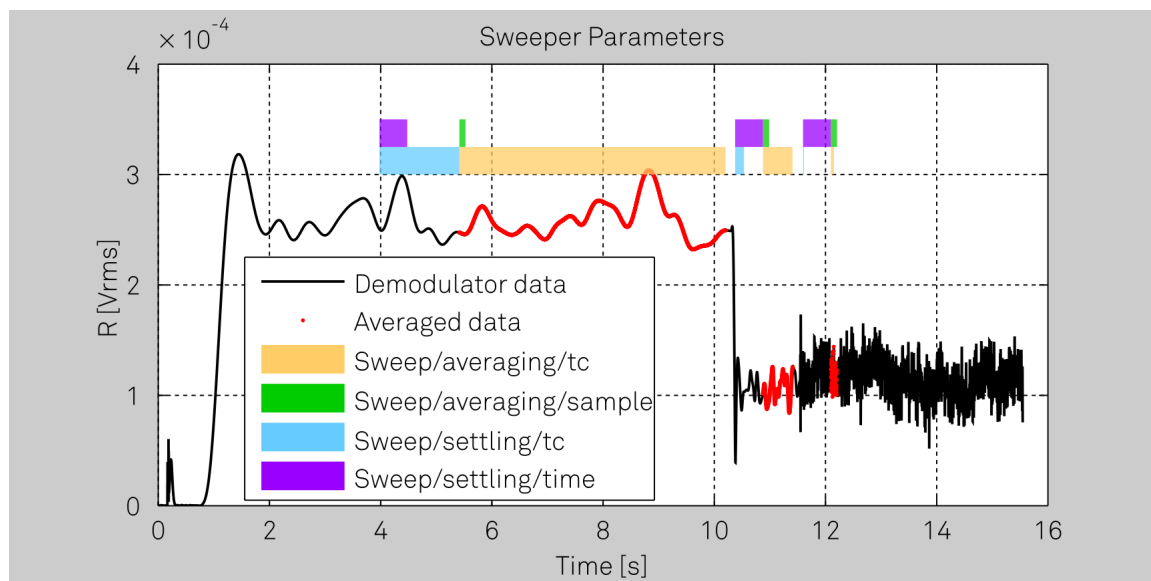


Figure 3.13. Plot demonstrating how the Sweeper records three measurement points from demodulator data when using automatic bandwidth control in a frequency sweep. Please see [An Explanation of Settling and Averaging Times in a Frequency Sweep](#), above, for a detailed explanation.

Average Power and Standard Deviation of the Measured Data

The Sweeper returns measurement data upon calling the Sweeper's `read()` function. This returns not only the averaged measured samples (e.g. `r`) but also their average power (`rpwr`) and standard deviation (`rstddev`). In order to obtain reliable values from this statistical data, please ensure that the `sweep/averaging` branch parameters are configured correctly. It's recommended to use at least a value of 12 for `sweep/averaging/sample` to ensure enough values are used to calculate the standard deviation and 5 for `sweep/averaging/tc` in order to prevent aliasing effects from influencing the result.

Table 3.26. Sweeper Parameters

Setting/Path	Type	Unit	Description
<code>sweep/device</code>	string	-	The device ID to perform the sweep on, e.g., <code>dev123</code> (compulsory parameter).
<code>sweep/gridnode</code>	string	Node	The device parameter (specified by node) to be swept, e.g., <code>"oscs/0/freq"</code> .
<code>sweep/start</code>	double	Many	The start value of the sweep parameter.
<code>sweep/stop</code>	double	Many	The stop value of the sweep parameter.
<code>sweep/samplecount</code>	uint64	-	The number of measurement points to set the sweep on.
<code>sweep/endless</code>	bool	-	Enable Endless mode; run the sweeper continuously.
<code>sweep/remainingtime</code>	double	Seconds	Read only: Reports the remaining time of the current sweep. A valid number is only displayed once the sweeper has been started. An undefined sweep time is indicated as NAN.
<code>sweep/averaging/sample</code>	uint64	Samples	Sets the number of data samples per sweeper parameter point that is considered in the measurement. The maximum of this value and <code>sweep/averaging/tc</code> is taken as the effective calculation time. See Figure 3.13 .
<code>sweep/averaging/tc</code>	double	Seconds	Sets the effective measurement time per sweeper parameter point that is considered in the measurement. The maximum between of this value and <code>sweep/averaging/sample</code> is taken as the effective calculation time. See Figure 3.13 .
<code>sweep/bandwidthcontrol</code>	uint64	-	Specify how the sweeper should specify the bandwidth of each measurement point, Automatic is recommended, in particular for logarithmic sweeps and assures the whole spectrum is covered. 0=Manual (the sweeper module leaves the demodulator bandwidth settings entirely untouched); 1=Fixed (use the value from <code>sweep/bandwidth</code>); 2=Automatic. Note, to use either Fixed or Manual mode, <code>sweep/bandwidth</code> must be set to a value > 0 (even though in manual mode it is ignored).

Setting/Path	Type	Unit	Description
sweep/ bandwidthoverlap	bool	-	If enabled the bandwidth of a sweep point may overlap with the frequency of neighboring sweep points. The effective bandwidth is only limited by the maximal bandwidth setting and omega suppression. As a result, the bandwidth is independent of the number of sweep points. For frequency response analysis bandwidth overlap should be enabled to achieve maximal sweep speed.
sweep/bandwidth	double	Hz	Defines the measurement bandwidth when using Fixed bandwidth mode (sweep/ bandwidthcontrol=1), and corresponds to the noise equivalent power bandwidth (NEP).
sweep/order	uint64	-	Defines the filter roll off to use in Fixed bandwidth mode (sweep/ bandwidthcontrol=1). Valid values are between 1 (6 dB/octave) and 8 (48 dB/octave).
sweep/maxbandwidth	double	Hz	Specifies the maximum bandwidth used when in Auto bandwidth mode (sweep/ bandwidthcontrol=2) (sweep/ bandwidthcontrol=2). The default is 1.25 MHz.
sweep/ omegasuppression	double	dB	Damping of omega and 2omega components when in Auto bandwidth mode (sweep/ bandwidthcontrol=2). Default is 40dB in favor of sweep speed. Use a higher value for strong offset values or 3omega measurement methods.
sweep/loopcount	uint64	-	The number of sweeps to perform.
sweep/phaseunwrap	bool	-	Enable unwrapping of slowly changing phase evolutions around the +/-180 degree boundary.
sweep/sincfilter	bool	-	Enables the sinc filter if the sweep frequency is below 50 Hz. This will improve the sweep speed at low frequencies as omega components do not need to be suppressed by the normal low pass filter.
sweep/scan	uint64	-	Selects the scanning type: 0=Sequential (incremental scanning from start to stop value, see Figure 3.14); 1=Binary (Non-sequential sweep continues increase of resolution over entire range, see Figure 3.16), 2=Bidirectional (Sequential sweep from Start to Stop value and back to Start again, Figure 3.15), 3=Reverse (reverse sequential scanning from stop to start value).
sweep/settling/time	double	Seconds	Minimum wait time in seconds between setting the new sweep parameter value and the start of the measurement. The maximum between this value and sweep/settling/

Setting/Path	Type	Unit	Description
			t_c is taken as effective settling time. See Figure 3.13 .
sweep/settling/inaccuracy	double	-	Demodulator filter settling inaccuracy defining the wait time between a sweep parameter change and recording of the next sweep point. The settling time is calculated as the time required to attain the specified remaining proportion [1e-13, 0.1] of an incoming step function. Typical inaccuracy values: 10m for highest sweep speed for large signals, 100u for precise amplitude measurements, 100n for precise noise measurements. Depending on the order of the demodulator filter the settling inaccuracy will define the number of filter time constants the sweeper has to wait. The maximum between this value and the settling time is taken as wait time until the next sweep point is recorded. The relationship between sweep/settling/inaccuracy and sweep/settling/ t_c is plotted in Figure 3.12 .
sweep/settling/ t_c	double	TC	Minimum wait time in factors of the time constant (TC) between setting the new sweep parameter value and the start of the measurement. This filter settling time is preferably configured via the sweep/settling/inaccuracy (see discussion in Section 3.11.1 and Figure 3.12). The maximum between this value and sweep/settling/time is taken as effective settling time. See Figure 3.13 .
sweep/xmapping	uint64	-	Selects the spacing of the grid used by sweep/gridnode (the sweep parameter): 0=linear and 1=logarithmic distribution of sweep parameter values.
sweep/historylength	uint64		Maximum number of entries stored in the measurement history.
sweep/clearhistory	bool	-	Remove all records from the history list.
sweep/directory	string	-	The directory to which sweeper measurements are saved to via save().
sweep/savepath	string	-	This parameter is deprecated, see sweep/directory.
sweep/fileformat	string	-	The format of the file for saving sweeper measurements. 0=Matlab, 1=CSV.

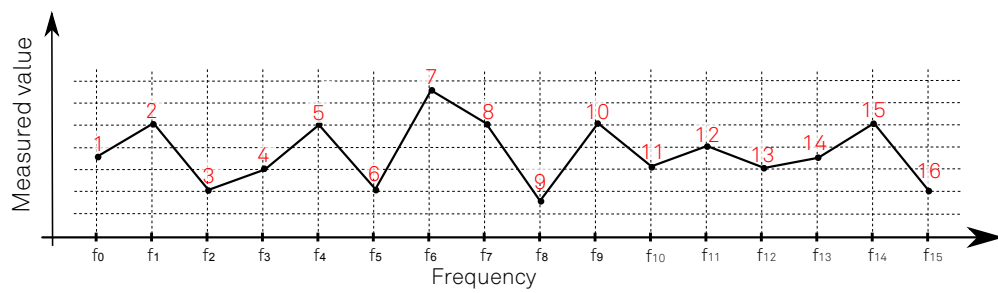


Figure 3.14. Sweeper scanning modes: Sequential ($\text{sweep}/\text{scan} = 0$).

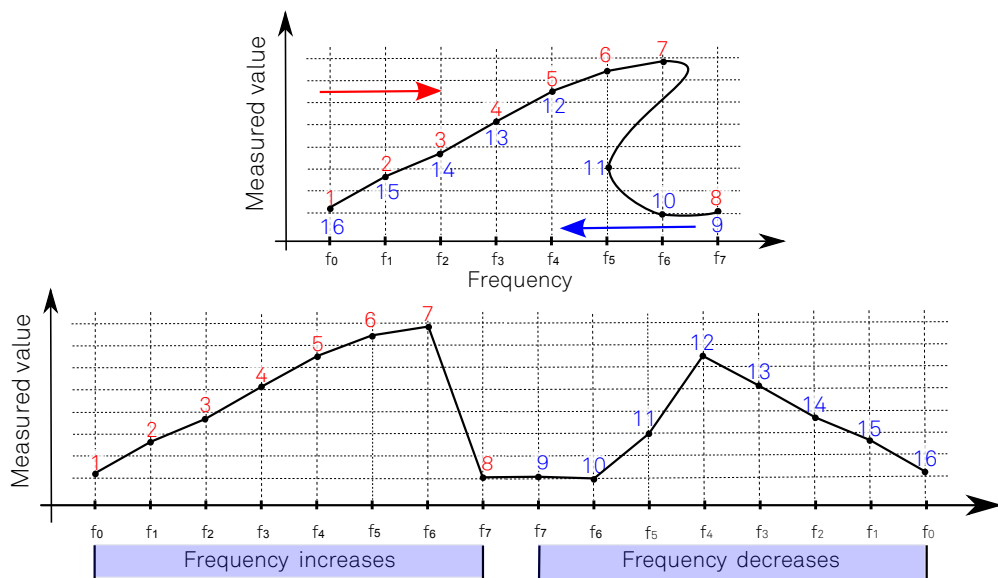


Figure 3.15. Sweeper scanning modes: Bidirectional ($\text{sweep}/\text{scan} = 2$).

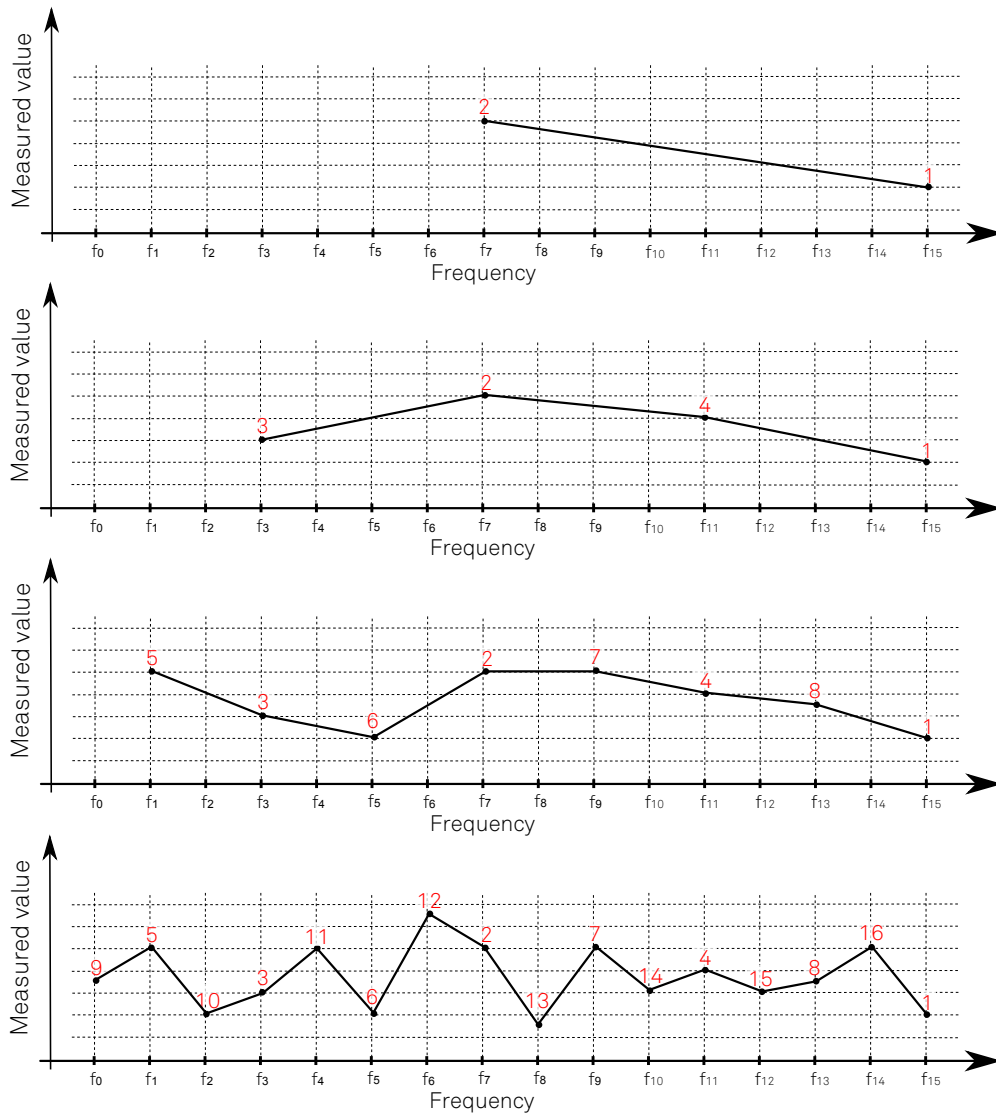


Figure 3.16. Sweeper scanning modes: Binary (sweep/scan = 1).

Table 3.27. Sweeper Output Values

Name	Type	Unit	Description
auxin0	double	Volts	Auxiliary Input 1 value.
auxin1	double	Volts	Auxiliary Input 2 value.
auxin0pwr	double	Volts ²	Average power of Auxiliary Input 1 value.
auxin1pwr	double	Volts ²	Average power of Auxiliary Input 2 value.
auxin0stddev	double	Volts	Standard deviation of Auxiliary Input 1 value.
auxin1stddev	double	Volts	Standard deviation of Auxiliary Input 2 value.
frequency	double	Hz	The oscillator frequency for each measurement point (for a frequency sweep this is the same as grid).
frequencypwr	double	Hz ²	Average power of the oscillator frequency.
frequenciesstddev	double	Hz	Standard deviation of the oscillator frequency.

Name	Type	Unit	Description
phase	double	Radians	Demodulator phase value.
phasestddev	double	Radians	Standard deviation of demodulator phase value (phase noise).
phaserpwr	double	Radians ²	Average power of demodulator phase value (phase noise).
r	double	VoltsRMS	Demodulator R value.
rstddev	double	VoltsRMS	Standard deviation of demodulator R value.
rpwr	double	Volts ²	Average power of demodulator x value.
x	double	Volts	Demodulator x value.
xstddev	double	Volts	Standard deviation of demodulator x value.
xpwr	double	Volts ²	Average power of demodulator x value.
y	double	Volts	Demodulator y value.
ystddev	double	Volts	Standard deviation of demodulator y value.
ypwr	double	Volts ²	Average power of demodulator y value.
bandwidth	double	Hz	Demodulator filter's bandwidth as calculated from sweep/tc (if performing a frequency sweep).
bandwidthmode	integer	-	The value of the sweep/bandwidthcontrol used for the sweep.
count	integer	-	The number of measurement points actually used by the sweeper when averaging the data. This depends on the values of the parameters in the sweep/averaging/branch.
grid	double	Many	Values of sweeping setting (frequency values at which demodulator samples where recorded).
flags	integer	-	Reserved for future use.
settling	double	Seconds	The waiting time for each measurement point.
samplecount	uint64	-	The number of swept measurement points (the value of sweep/samplecount).
sampleformat	integer	-	Reserved for future use.
sweepmode	integer	-	The value of the sweep/scan used for the sweep.
tc	double	Seconds	Demodulator's filter time constant as set for each measurement point.
tcmeas	double	Seconds	Reserved for future use.
timestamp	uint64	Ticks	A timestamp that gets updated each time a new measurement point has been recorded by the sweeper (divide by the device's clockbase to obtain seconds). It is not part of the sweeper's measurement data and only relevant for intermediate reads of sweeper data (before the current sweep has finished).

Name	Type	Unit	Description
setttimestamp	uint64	Ticks	The timestamp at which we verify that the frequency for the current measurement point was set on the device (by reading back demodulator data).
nexttimestamp	uint64	Ticks	The timestamp at which we can obtain the data for that measurement point, i.e., <code>nexttimestamp - setttimestamp</code> corresponds roughly to the demodulator filter settling time.

3.12. Software Trigger (Recorder) Module

In LabOne Releases prior to 17.12, the Recorder Module corresponds to the Software Trigger Tab of the LabOne User Interface. It allows the user to record bursts of instrument data based upon pre-defined trigger criteria similar to that of a laboratory oscilloscope, see [Figure 3.17](#) for an example. The types of trigger available are listed in [Table 3.28](#).

Future Deprecation Warning

In LabOne Release 17.12 and subsequent releases the Recorder Module has been superseded by the Data Acquisition Module. We strongly recommend using the Data Acquisition Module instead of the Recorder Module for time (and frequency) domain data acquisition. See [Section 3.5](#) for a description of the [Data Acquisition Module](#). In particular, existing users of the Recorder module can use the guide in [Section 3.5.4, Migrating a SW Trigger program to the DAQ Module](#).

Table 3.28. Overview of the trigger types available in the Software Trigger Module.

Trigger Type	Description	trigger/N/type
Manual	For simple recording.	0
Edge	Edge trigger with level hysteresis and noise rejection, see Figure 3.18 .	1
Digital	Digital trigger with bit masking.	2
Pulse	Pulse width trigger with level hysteresis and noise reduction, see Figure 3.19 and Figure 3.20 .	3
Tracking (edge or pulse)	Level tracking trigger to compensate signal drift, see Figure 3.21 .	4
Hardware Trigger	UHFLI and MFLI only. Trigger on one of the instrument's hardware trigger channels.	6

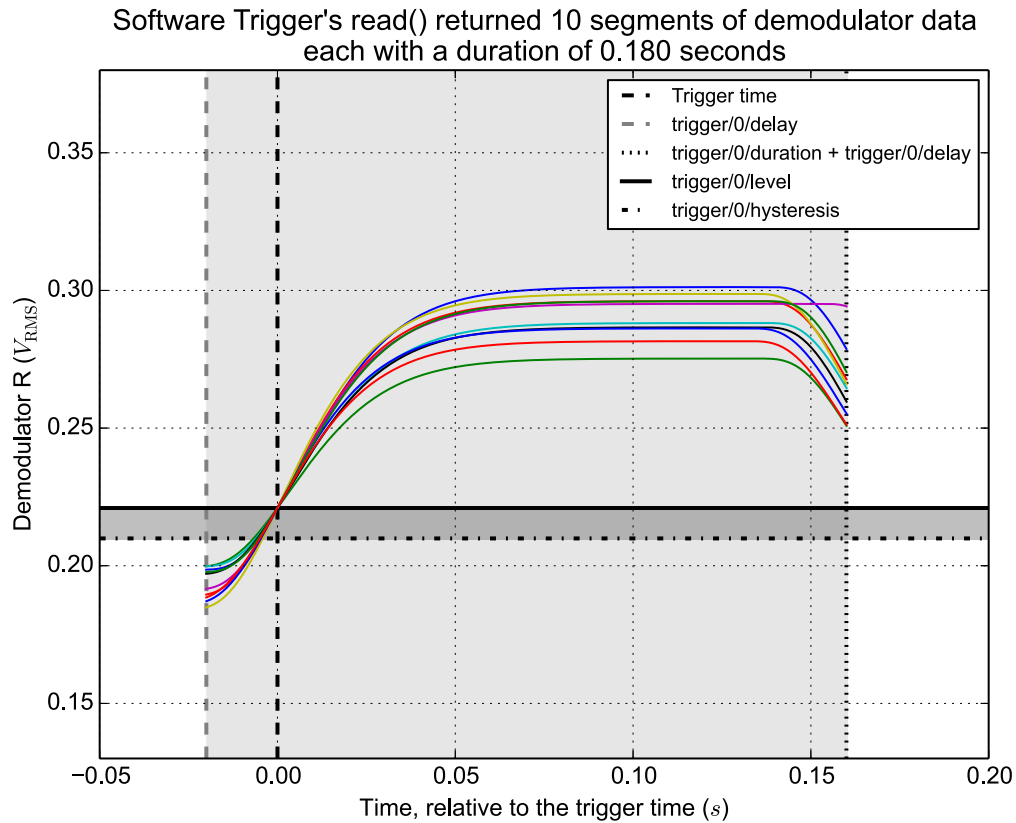


Figure 3.17. The plot produced by `example_swtrigger_edge.py`, an example distributed with the LabOne Python API. The plot shows 10 bursts of data from a single demodulator; each burst was recorded when the demodulator's R value exceeded a specified threshold using a positive edge trigger. See [Section 5.2.3](#) for help getting started with the Python examples.

See [Table 3.29](#) for the input parameters to configure the Software Trigger's Module. Note that some parameters effect all triggers, e.g., `trigger/endless`, whereas some are configured on a per-trigger basis, e.g., `trigger/N/duration`, where N is the index of the trigger, starting at zero. The data output when using the Software Trigger's `read` command has the same format as returned by `ziCore's poll` command.

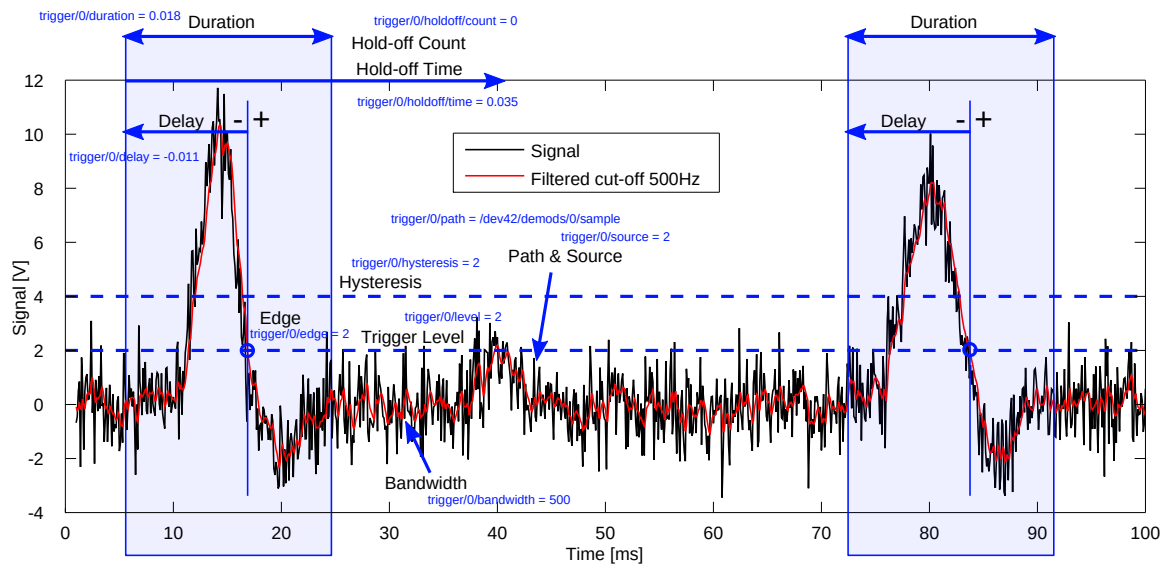


Figure 3.18. Explanation of the Software Trigger Module's parameters for an Edge Trigger.

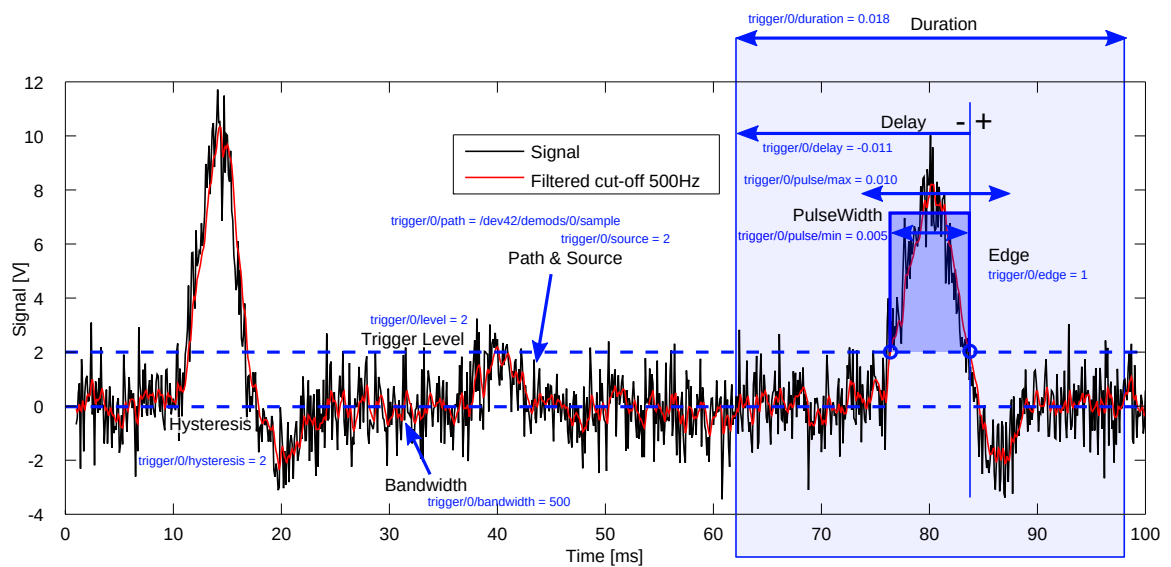


Figure 3.19. Explanation of the Software Trigger Module's parameters for a positive Pulse Trigger.

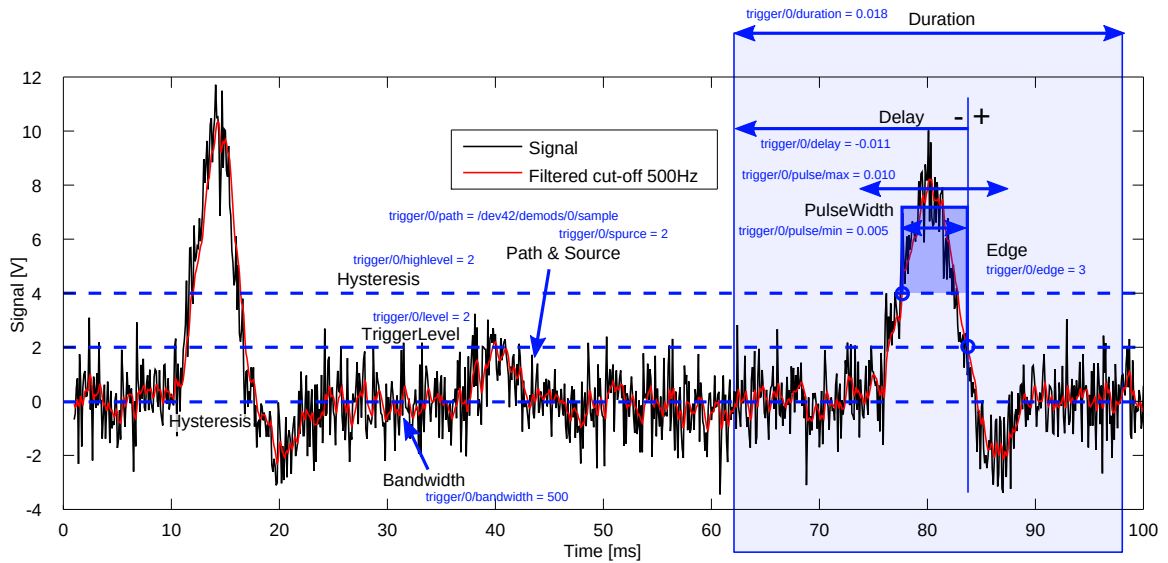


Figure 3.20. Explanation of the Software Trigger parameters for a positive or negative Pulse Trigger.

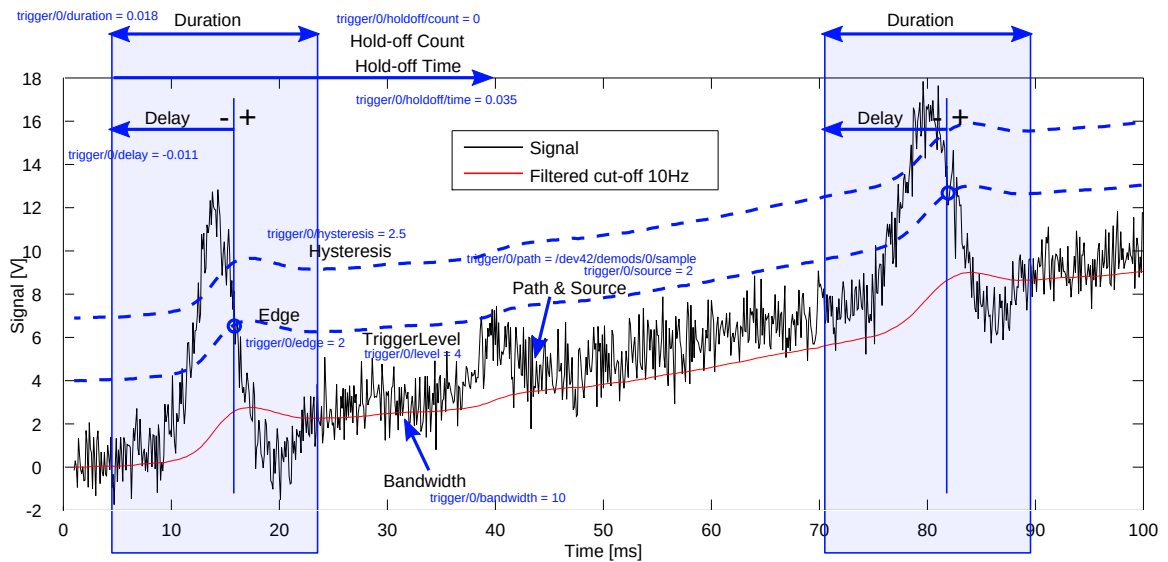


Figure 3.21. Explanation of the Software Trigger Module's parameters for a Tracking Trigger.

3.12.1. Determining the Trigger Level automatically

The SW Trigger Module can calculate the `trigger/0/level` and `trigger/0/hysteresis` parameters based on the current input signal for edge, pulse, tracking edge and tracking pulse trigger types. This is particularly useful when using a tracking trigger, where the trigger level is relative to the output of the low-pass filter tracking the input signal's average (see [Figure 3.21](#)). In the LabOne User Interface this functionality corresponds to the "Find" button in the Settings sub-tab of the SW Trigger Tab.

This functionality is activated via API by setting the `trigger/0/findlevel` parameter to 1. This is a single-shot calculation of the level and hysteresis parameters, meaning that it is performed only once, not continually. The SW Trigger monitors the input signal for a duration of 0.1 seconds and sets the level parameter to the average of the largest and the smallest values detected in the signal and the hysteresis to 10% of the difference between largest and smallest values. When

the SW Trigger has finished its calculation of the level and hysteresis parameters it sets the value of the `trigger/0/findlevel` parameter to 0 and writes the values to the `trigger/0/level` and `trigger/0/hysteresis` parameters. Note that the calculation is only performed if the SW Trigger Module is currently running, i.e., after `execute()` has been called. See [Example 3.4](#) for Python code demonstrating how to use this behaviour.

```
# Start the Software Trigger's thread. Ready to record triggers.
trigger.execute()
# Tell the SW Trigger to determine the trigger level.
trigger.set('trigger/0/findlevel', 1)
time.sleep(0.1) # Ensure findlevel has been set before continuing.
trigger_params = trigger.get('trigger/*', True)
timeout = 10 # [s]
t0 = time.time()
# Wait until the levels have been found (when findlevel is set to 0).
while trigger_params['/0/findlevel'] == 1:
    time.sleep(0.05)
    trigger_params = trigger.get('trigger/*', True)
    if time.time() - t0 > timeout:
        trigger.finish()
        trigger.clear()
        raise RuntimeError("SW Trigger didn't find trigger level after %.3f seconds."
% timeout)
    print("Level: {}".format(trigger_params['/0/level'][0]))
    print("Hysteresis: {}".format(trigger_params['/0/hysteresis'][0]))
```

Example 3.4. Python code demonstrating how to use the `trigger/0/findlevel` parameter. Taken from the Python example `example_swtrigger_grid`.

3.12.2. Using the SW Trigger with a Digital Trigger

To use the SW Trigger with a digital trigger, it must be configured to use a digital trigger type (by setting `trigger/0/type` to 2) and to use the output value of the instrument's DIO port as its trigger source. This is achieved by setting `trigger/0/triggernode` to the device node `/devn/demods/m/sample.dio`. It is important to be aware that the SW Trigger takes its value for the DIO output from the demodulator sample field bits, not from a node in the `/devn/dios/` branch. As such, the specified demodulator must be enabled and an appropriate transfer rate configured that meets the required trigger resolution (the SW Trigger can only resolve triggers at the resolution of $1/(\text{devn/demods/m/rate})$; it is not possible to interpolate a digital signal to improve trigger resolution and if the incoming trigger pulse on the DIO port is shorter than this resolution, it may be missed).

The Digital Trigger allows not only the trigger bits (`trigger/0/bits`) to be specified but also a bit mask (`trigger/0/mask`) in order to allow an arbitrary selection of DIO pins to supply the trigger signal. When a positive, respectively, negative edge trigger is used, all of these selected pins must become high, respectively low. The bit mask is applied as following. For positive edge triggering (`trigger/0/edge` set to value 1), the SW Trigger is triggered when the following equality holds for the DIO value:

```
(/devn/demods/m/sample.dio BITAND trigger/0/mask) == (trigger/0/bits BITAND
trigger/0/mask)
```

and this equality has not been met for the previous value in time (the previous sample) of `/devn/demods/m/sample.dio`. For negative edge triggering (`trigger/0/edge` set to value 2), the SW Trigger is triggered when the following inequality holds for the current DIO value:

```
(/devn/demods/m/sample.dio BITAND trigger/0/mask) != (trigger/0/bits BITAND
trigger/0/mask)
```

and this inequality was not met (there was equality) for the previous value of the DIO value.

3.12.3. The Software Trigger's "retrigger" functionality

If the parameter `trigger/0/retrigger` is enabled (set to 1), then the length of the current trigger's data segment is extended if another trigger event occurs with the configured `trigger/0/duration`. In other words, the returned data will have a length greater than `trigger/0/duration` and contain multiple trigger events. The maximum size of the returned data corresponds to the value of `trigger/buffersize`; it is not possible to retrigger beyond this time and a new trigger data segment will be started if required. Note, if a longer `trigger/buffersize` is desired it should be configured after setting the `trigger/0/duration` parameter to avoid an automatic adjustment of `trigger/buffersize` by the Software Trigger.

Table 3.29. Software Trigger Input Parameters.

Setting/Path	Type	Unit	Description
<code>trigger/device</code>	string	-	The device ID to execute the software trigger, e.g., <code>dev123</code> (compulsory parameter).
<code>trigger/buffersize</code>	double	Seconds	Set the buffer size of the trigger object. The recommended buffer size is $2 * \text{trigger}/N / \text{duration}$.
<code>trigger/flags</code>	uint64	-	Define the SW Trigger's behaviour if sampleloss is encountered: Fill holes (<code>=0x01</code>), align data that contains a timestamp (<code>=0x02</code>), throw <code>EOFError</code> if sampleloss is detected.
<code>trigger/endless</code>	uint64	-	Enable endless triggering 1=enable; 0=disable.
<code>trigger/forcetrigger</code>	uint64	-	Force a trigger.
<code>trigger/filename</code>	string	-	This parameter is deprecated. If specified, i.e. not empty, it enables automatic saving of data in single triggering mode (<code>trigger/endless = 0</code>).
<code>trigger/savepath</code>	string	-	The directory where files are saved when saving data.
<code>trigger/fileformat</code>	string	-	The format of the file for saving data. 0=Matlab, 1=CSV.
<code>trigger/historylength</code>	uint64	-	Maximum number of entries stored in the measurement history.
<code>trigger/clearhistory</code>	uint64	-	Clear the measurement history
<code>trigger/triggered</code>	uint64	-	Has the software trigger triggered? 1=Yes, 0=No (read only).
<code>trigger/N/bandwidth</code>	double	Hz	Only for Tracking Triggers. The bandwidth used in the calculation of the exponential running average of the source signal.
<code>trigger/N/bitmask</code>	uint64	-	Only for Digital triggers. Specify the bitmask used with <code>trigger/N/bits</code> . The trigger value is bits AND bit mask (bitwise).
<code>trigger/N/bits</code>	uint64	-	Only for Digital triggers. Specify the bits used for the Digital trigger value. The trigger value is bits AND bit mask (bitwise)

Setting/Path	Type	Unit	Description
trigger/N/count	uint64	-	The number of triggers to save.
trigger/N/delay	uint64	Seconds	The amount of time to record data before the trigger was activated, Delay: Time delay of trigger frame position (left side) relative to the trigger edge. For delays smaller than 0, trigger edge inside trigger frame (pre trigger). For delays greater than 0, trigger edge before trigger frame (post trigger), see Figure 3.18 .
trigger/N/duration	double	Seconds	The length of time to record data for, see Figure 3.18 .
trigger/N/edge	uint64	-	Define on which signal edge to trigger. Triggers when the trigger input signal crosses the trigger level from either low to high (edge=1), high to low (edge=2) or both (edge=3). Used for Trigger Type edge, pulse, tracking edge and tracking pulse. In the case of pulse trigger, the value specifies a positive (edge=1) or negative (edge=2) pulse relative to the trigger level (edge=3 specifies either positive or negative).
trigger/N/findlevel	uint64	-	Automatically find the value of trigger/N/level based on the current signal value.
trigger/N/level	uint64	Many	Specify the main trigger level value.
trigger/N/holdoff/count	uint64	-	The holdoff count, the number of skipped triggers until the next trigger is recorded again.
trigger/N/holdoff/time	double	Seconds	The holdoff time, the amount of time until the next trigger is recorded again. A hold off time smaller than @trigger/0/duration@ will produce overlapping trigger frames.
trigger/N/hysteresis	double	Many	Specify the hysteresis value (the trigger is re-armed after the signal exceeds trigger/N/level and then falls below trigger/N/hysteresis, if using positive edge).
trigger/N/pulse/max	double	-	Only for Pulse triggers: The maximum pulse width to trigger on. See Figure 3.19 .
trigger/N/pulse/min	double	-	Only for Pulse triggers: The minimum pulse width to trigger on. See Figure 3.19 .
trigger/N/retrigger	uint64	-	1=enable, 0=disable. Enable to allow re-triggering within one trigger duration. If enabled continue recording data in one segment if another trigger comes within the previous trigger's duration. If disabled the triggers will be recorded as separate events.
trigger/N/triggernode	string	-	Path and signal of the node that should be used for triggering, separated by a dot (.), e.g. /devN/demods/0/sample.x. SAMPLE.X Demodulator X value SAMPLE.Y Demodulator Y value

Setting/Path	Type	Unit	Description
			<code>SAMPLE.R</code> Demodulator Magnitude <code>SAMPLE.THETA</code> Demodulator Phase <code>SAMPLE.AUXIN0</code> Auxiliary Input 1 value <code>SAMPLE.AUXIN1</code> Auxiliary Input 2 value <code>SAMPLE.DIO</code> Digital I/O value Over HW Trigger paths may also be specified (device-class dependent). Overrides values from <code>trigger/0/path</code> and <code>trigger/0/source</code>.
<code>trigger/N/type</code>	uint64	-	The trigger type, see Table 3.28
<code>trigger/0/grid/mode</code>	int	-	Enable grid mode. In Grid Mode a matrix instead of a vector is returned by <code>read()</code> . Each trigger becomes a row in the matrix and each trigger's data is interpolated onto a new grid defined by the number of columns: 0: Disable, 1: Enable grid mode with nearest neighbour interpolation, 2: Enable grid mode with linear interpolation.
<code>trigger/0/grid/repetitions</code>	int	-	The number of times to perform <code>trigger/0/grid/operation</code> on the data in one grid.
<code>trigger/0/grid/operation</code>	int	-	If <code>trigger/0/grid/repetitions</code> is greater than 1, either replace or average the data in the grid's matrix.
<code>trigger/0/grid/cols</code>	int	-	Specify the number of columns in the grid's matrix. The data from each row is interpolated onto a grid with the specified number of columns.
<code>trigger/0/grid/rows</code>	int	-	Specify the number of rows in the grid's matrix. Each row is the data recorded from one trigger interpolated onto the columns.
<code>trigger/0/grid/direction</code>	int	-	The direction to organize data in the grid's matrix: 0: Forward. 1: Reverse. 2: Bidirectional. Forward - the data in each row is ordered chronologically, e.g., the first data point in each row corresponds to the first timestamp in the trigger data. Reverse - the data in each row is ordered reverse chronologically, e.g., the first data point in each row corresponds to the last timestamp in the trigger data. Bidirectional - the ordering of the data alternates between Forward and Backward ordering from row-to-row, the first row is Forward ordered.
<code>trigger/N/path</code>	string	-	This parameter is deprecated, see the <code>trigger/N/triggernode</code> parameter.
<code>trigger/N/source</code>	uint64	-	This parameter is deprecated, see the <code>trigger/N/triggernode</code> parameter.
<code>trigger/N/hwtrigsources</code>	uint64	-	This parameter is deprecated, see the <code>trigger/N/triggernode</code> parameter.

Level and Hysteresis Settings with a Pulse Trigger

For the pulse trigger type, there is a subtle difference between the way the trigger level and the hysteresis are used for positive/negative pulse triggering (trigger/N/edge= 1 or 2) and both (trigger/N/edge= 3). The difference can be seen in [Figure 3.19](#) and [Figure 3.20](#).

3.13. Spectrum Analyzer Module

In LabOne Releases prior to 17.12, the Spectrum Analyzer Module allows access to the functionality available in the Spectrum Analyzer Tab of the LabOne User Interface from an API. It's a measurement tool that performs Fast Fourier Transforms (FFT) on demodulator output.

Future Deprecation Warning

In LabOne Release 17.12 and subsequent releases the Spectrum Analyzer (zoomFFT) Module has been superseded by the Data Acquisition Module. We strongly recommend using the Data Acquisition Module instead of the Recorder Module for frequency (and time) domain data acquisition. See [Section 3.5](#) for a description of the [Data Acquisition Module](#). In particular, existing users of the Recorder module can use the guide in [Section 3.5.5, Migrating a zoomFFT program to the DAQ Module](#).

To get started please refer to the Spectrum Analyzer Tab in the relevant instrument-specific User Manual. An example is available in the each of the LabVIEW, Matlab, Python or .NET APIs.

See [Table 3.30](#) for the input parameters to configure the Spectrum Module and [Table 3.31](#) for a description of the Spectrum Module outputs.

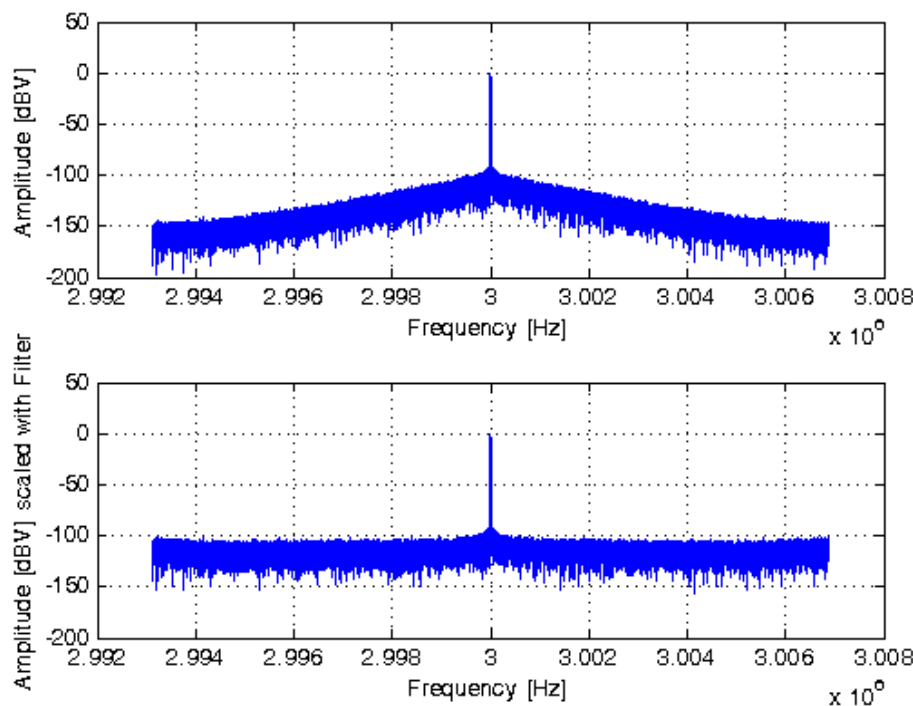


Figure 3.22. The FFT result from the LabOne Matlab API Spectrum Module example.

Table 3.30. Spectrum Module Input Parameters

Setting/Path	Type	Unit	Description
zoomFFT/device	string	-	The device ID that is being used for the measurement; the demodulator data is taken from this device. For example, dev123 (compulsory parameter).

Setting/Path	Type	Unit	Description
zoomFFT/absolute	bool	-	When enabled, shifts the frequencies in the output grid so that the center frequency corresponds to the demodulation frequency. If not enabled, the center frequency is 0 Hz.
zoomFFT/bit	uint64	-	Number of lines of the FFT spectrum (powers of 2). A higher value increases the frequency resolution of the spectrum.
zoomFFT/endless	bool	-	Enable Endless mode; Run the FFT spectrum analysis continuously.
zoomFFT/loopcount	uint64	-	The number of FFTs to perform if not running in Endless mode.
zoomFFT/mode	uint64	-	Select the source signal for the FFT. 0=FFT ($x+iy$) Complex FFT of the demodulator result. 1=FFT (R) FFT of the demodulator amplitude $\sqrt{x^2 + y^2}$. The FFT is single-sided as performed on real data. 2=FFT (phase) FFT of the demodulator phase $\text{atan2}(y, x)$. The FFT is single-sided as performed on real data. 3=FFT (f) FFT of the oscillator frequency of the selected demodulator. This mode is only interesting if the oscillator is controlled by a PID / PLL controller. The FFT is single-sided as performed on real data. 4=FFT ($d\theta/dt$) / (2π) FFT of the demodulator phase derivative. This value is equivalent to the frequency noise observed on the demodulated signal. The FFT is single-sided as performed on real data.

Setting/Path	Type	Unit	Description
zoomFFT/overlap	double	-	Overlap of the demodulator data used for the FFT. Use 0 for no overlap and 0.99 for maximal overlap.
zoomFFT/settling/tc	double	TC	Minimum wait time in factors of the demodulator time constant (TC) before starting the measurement. The maximum of this value and zoomFFT/settling/time is taken as the effective settling time.
zoomFFT/settling/time	double	s	Minimum wait time in seconds before starting the measurement. The maximum of this value and zoomFFT/settling/tc is taken as effective settling time.
zoomFFT/window	uint64	-	The type of FFT window to use. 0=Rectangular, 1=Hann, 2=Hamming, 3=Blackman Harris.

Table 3.31. Spectrum Module Output Values

Name	Type	Unit	Description
x	double	VoltsRMS	The real part, x, of the complex FFT result.
y	double	VoltsRMS	The imaginary part, y, of the complex FFT result.
r	double	VoltsRMS	The absolute value, $R = \text{Abs}(X + iY)$, of the complex FFT result.
timestamp	uint64	Ticks	Demodulator timestamp of the measurement (divide by the device's clockbase to obtain seconds).
center	double	Hz	The center frequency (corresponds to the demodulation frequency) of the spectrum.
rate	double	1/s	Sampling rate of the demodulator.
filter	double	-	The absolute values of the demodulator's low-pass filter transfer function for each grid point. The FFT result, x, y or R, may be divided by this value to obtain a filter compensated spectrum.
bandwidth	double	Hz	The noise-equivalent power bandwidth of the demodulator
grid	double	Hz	The frequency grid.
nenbw	double	-	The normalized equivalent noise bandwidth. If calculating spectral densities multiply your spectrum by this value to correct for windowing effects.
resolution	double	Hz	FFT resolution: Spectral resolution defined by the reciprocal acquisition time (sampling rate/number of samples recorded).
aliasingreject	double	dB	The damping of the demodulator present at the border of the spectrum.

Chapter 4. Matlab Programming

The MathWorks' numerical computing environment [Matlab®](#) has powerful tools for data analysis and visualization that can be used to create graphical user interfaces or automatically generate reports of experimental results in various formats. LabOne's Matlab API, also known as `ziDAQ`, "Zurich Instruments Data Acquisition", enables the user to stream data from their instrument directly into Matlab allowing them to take full advantage of this powerful environment.

This chapter aims to help you get started using Zurich Instruments LabOne's Matlab API, `ziDAQ`, to control your instrument, please refer to:

- [Section 4.1](#) for help [Installing the LabOne Matlab API](#).
- [Section 4.2](#) for help [Getting Started with the LabOne Matlab API](#) and [Running the Examples](#).
- [Section 4.3](#) for some [LabOne Matlab API Tips and Tricks](#).
- [Section 4.4](#) for help [Troubleshooting the LabOne Matlab API](#).
- [Section 4.5](#) for [LabOne Matlab API \(ziDAQ\) Command Reference](#).

Note

This section and the provided examples are not intended to be a Matlab tutorial. See either MathWorks' online [Documentation Center](#) or one of the many online resources, for example, the [Matlab Programming Wikibook](#) for help to get started programming with Matlab.

4.1. Installing the LabOne Matlab API

4.1.1. Requirements

One of the following platforms and Matlab versions (with valid license) is required to use the LabOne Matlab API:

1. 32 or 64-bit Windows with Matlab R2009b or newer.
2. 64-bit Linux with Matlab R2018a or newer.
3. 64-bit Mac OS X and Matlab R2013b or newer.

The LabOne Matlab API `ziDAQ` is included in a standard LabOne installation and is also available as a separate package (see below, [Separate Matlab Package](#)). No installation as such is required, only a few configuration steps must be performed to use `ziDAQ` in Matlab. Both the main LabOne installer and the separate LabOne Matlab API package are available from Zurich Instruments' [download page](#).

Separate Matlab Package

The separate Matlab API package should be used if you would like to:

1. Use the Matlab API on Mac OS X (the main LabOne installer is not available for Mac OS X).
2. Use the Matlab API to work with an instrument remotely (i.e., on a separate PC from where the Data Server is running) and you do not require a full LabOne installation. This is the case, for example, with MF Instruments.
3. Use the Matlab API on a PC where you do not have administrator rights.

Mac OS X

Since the Data Server is not available for use on Mac OS X, using the Matlab API on OS X requires a Data Server running remotely on either an instrument or a separate Windows or Linux PC. If the Data Server is running on a PC, please ensure that it's configured to accept connections from other PCs in the network as described in [Section 1.4.1](#).

4.1.2. Windows, Linux or Mac

No additional installation steps are required to use `ziDAQ` on either Windows, Linux or Mac; it's only necessary to add the folder containing LabOne's Matlab Driver to Matlab's search path. This is done as following:

1. Start Matlab and either set the "Current Folder" (current working directory) to the Matlab API folder in your LabOne installation or the extracted zip archive of the separate Matlab API package (see above, [Separate Matlab Package](#)) as appropriate.

If using a LabOne installation on Windows this is typically:

```
C:\Program Files\Zurich Instruments\LabOne\API\MATLAB2012\
```

and on Linux this is the location where you unpacked the LabOne `.tar.gz` file:

```
[PATH]/LabOne64/API/MATLAB2012/
```

On Mac please use the [Separate Matlab Package](#).

2. In the Matlab Command Window, run the Matlab script `ziAddPath` located in the `MATLAB2012` directory:

```
>> ziAddPath;
```

On Windows (similar for Linux and Mac) you should see the following output in Matlab's Command Window:

```
Added ziDAQ's Driver, Utilities and Examples directories to Matlab's path  
for this session.
```

To make this configuration persistent across Matlab sessions either:

1. Run the `'pathtool'` command in the Matlab Command Window and add the following paths WITH SUBFOLDERS to the Matlab search path:

```
C:\Program Files\Zurich Instruments\LabOne\API\MATLAB2012\
```

or

2. Add the following line to your Matlab `startup.m` file:

```
run('C:\Program Files\Zurich Instruments\LabOne\API\MATLAB2012\ziAddPath');
```

This is sufficient configuration if you would only like to use `ziDAQ` in the current Matlab session.

3. To make this configuration persistent between Matlab sessions do either one of the next two steps (as also indicated by the output of `ziAddPath`):
 - a. Run the `pathtool` and click "Add with Subfolders". Browse to the "MATLAB2012" directory that was located above in Step 1 and click "OK".
 - b. Edit your `startup.m` to contain the line indicated in the output from Step 2 above. For more help on Matlab's `startup.m` file, type the following in Matlab's Command Window:

```
>> docsearch('startup.m')
```

4. Verify your Matlab configuration as described in [Section 4.1.3](#).

4.1.3. Verifying Successful Matlab Configuration

In order to verify that Matlab is correctly configured to use `ziDAQ` please perform the following steps:

1. Ensure that the correct Data Server is running for your HDAWG, HF2 or UHF Instrument (the Data Server on MF Instruments starts when the device is powered on). The quickest way to check is to start the User Interface for your device, see [Section 1.2](#) for more details.
2. Proceed either of the following two ways:
 - a. The easiest way to verify correct configuration is run one of the Matlab API's examples. In the Matlab command Window run, for example, `example_poll` with your device ID as the input argument:

```
>> example_poll('dev123'); % Replace with your device ID.
```

If this fails, please try issuing the `connect` command, as described in the next method.

- b. If a device is not currently available, correct Matlab API configuration can be checked by initializing a API session to the Data Server without device communication.

An API session with the Data Server is created using `ziDAQ`'s `connect` (the port specifies which Data Server to connect to on the localhost) cf. [Section 1.4.1](#)). In the Matlab command window type one of the following:

- `>> ziDAQ('connect', 'localhost', 8005) % 8005 for HF2 Series`
- `>> ziDAQ('connect', 'localhost', 8004, 6) % 8004 for HDAWG, UHFLLI`
- `>> ziDAQ('connect', mf-hostname, 8004, 6) % 8004 for MFLI (see below)`

Note, using `'localhost'` above assumes that the Data Server is running on the same computer from which you are using Matlab. See [Section 1.4.1](#) for information about ports and hostnames when connecting to the Data Server. For MFLI instruments the hostname/IP address of the MFLI instrument must be provided (the value of `mf-hostname`), see [Section 1.4.1](#) and the Getting Started chapter of the MFLI User Manual for more information.

3. If no error is reported then Matlab is correctly configured to use `ziDAQ` - congratulations! Otherwise, please try the steps listed in [Troubleshooting the LabOne Matlab API](#).

4.2. Getting Started with the LabOne Matlab API

This section introduces the user to the LabOne Matlab API.

4.2.1. Contents of the LabOne Matlab API

Alongside the driver for interfacing with your Zurich Instruments device, the LabOne Matlab API includes many files for documentation, utility functions and examples. See the `Contents.m` file located in a LabOne Matlab API directory (see Step 1 in [Section 4.1.2](#) for its typical location) for a description of the API's sub-folders and files. Run the command:

```
>> doc('Contents')
```

in the Matlab Command Window in the LabOne Matlab API directory to access the following contents interactively in Matlab.

```
% ziDAQ : The LabOne Matlab API for interfacing with Zurich Instruments Devices
%
% FILES
%   ziAddPath - add the LabOne Matlab API drivers, utilities and examples to
%               Matlab's Search Path for the current session
%   README.txt - a README briefly describing how to get started with ziDAQ
%
% DIRECTORIES
%   Driver/    - contains Matlab driver for interfacing with Zurich Instruments
%               devices
%   Utils/     - contains some utility functions for common tasks
%   Examples/  - contains examples for performing measurements on Zurich
%               Instruments devices
%
% DRIVER
%   Driver/ziDAQ.m          - ziDAQ command reference documentation.
%   Driver/ziDAQ.mex*      - ziDAQ API driver
%
% UTILS
%   ziAPIServerVersionCheck - check the versions of API and Data Server match
%   ziAutoConnect          - Create a connection to a Zurich Instruments
%                           server (Deprecated: See ziCreateAPISession).
%   ziAutoDetect           - Return the ID of a connected device (if only one
%                           device is connected)
%   ziBW2TC                - Convert demodulator 3dB bandwidth to timeconstant
%   ziCheckPathInData      - Check whether a node is present in data and non-empty
%   ziCreateAPISession     - Create an API session for the specified device with
%                           the correct Data Server.
%   ziDevices              - Return a cell array of connected Zurich Instruments
%                           devices
%   ziDisableEverything    - Disable all functional units and streaming nodes
%   ziGetDefaultSettingsPath - Get the default settings file path from the
%                           ziDeviceSettings ziCore module
%   ziGetDefaultSigoutMixerChannel - return the default output mixer channel
%   ziLoadSettings         - Load instrument settings from file
%   ziSaveSettings         - Save instrument settings to file
%   ziSiginAutorange       - Activate the device's autorange functionality
%   ziSystemtime2Matlabtime - Convert the LabOne system time to Matlab time
%   ziTC2BW                - Convert demodulator timeconstants to 3 dB Bandwidth
%
% EXAMPLES/COMMON - Examples that will run on most Zurich Instruments Devices
%   example_connect        - A simple example to demonstrate how to
%                           connect to a Zurich Instruments device
%   example_connect_config - Connect to and configure a Zurich
%                           Instruments device
%   example_save_device_settings_simple - Save and load device settings
%                                       synchronously using ziDAQ's utility
%                                       functions
```

```
% example_save_device_settings_expert - Save and load device settings
%                                     asynchronously with ziDAQ's
%                                     devicesettings module
% example_data_acquisition_continuous - Record data continuously using the
%                                     DAQ Module
% example_data_acquisition_edge       - Record bursts of demodulator data upon
%                                     a rising edge using the DAQ Module
% example_data_acquisition_fft        - Record the FFT of demodulator data
%                                     using the DAQ Module
% example_data_acquisition_grid       - Record bursts of demodulator data
%                                     and align the data in a 2D array.
% example_data_acquisition_grid_average - Record bursts of demodulator data
%                                     and align and average the data
%                                     row-wise in a 2D array.
% example_sweeper                    - Perform a frequency sweep using ziDAQ's
%                                     sweep module
% example_sweeper_rstddev_fixedbw    - Perform a frequency sweep plotting the
%                                     stddev in demodulator output R using
%                                     ziDAQ's sweep module
% example_sweeper_two_demods         - Perform a frequency sweep saving data
%                                     from 2 demodulators using ziDAQ's sweep
%                                     module
% example_poll                       - Record demodulator data using
%                                     ziDAQServer's synchronous poll function
% example_poll_impedance             - Record impedance data using
%                                     ziDAQServer's synchronous poll function
% example_scope                      - Record scope data using ziDAQServer's
%                                     synchronous poll function
% example_pid_advisor_pll            - Setup and optimize a PID for internal
%                                     PLL mode
% example_autoranging_impedance      - Demonstrate how to perform a manually
%                                     triggered autoranging for impedance
%                                     while working in manual range mode
% example_multidevice_data_acquisition - Acquire data from 2 synchronized
%                                     devices using the DAQ Module
% example_multidevice_sweep          - Perform a sweep on 2 synchronized
%                                     devices using the Sweeper Module
%
% EXAMPLES/HDAWG - Examples specific to the HDAWG Series
% hdaqg_example_awg                 - demonstrate different methods to
%                                     create waveforms and compile and
%                                     upload a SEQC program to the AWG
%
% EXAMPLES/HF2 - Examples specific to the HF2 Series
% hf2_example_autorange             - determine and set an appropriate range
%                                     for a sign channel
% hf2_example_poll_hardware_trigger - Poll demodulator data in combination
%                                     with a HW trigger
% hf2_example_scope                 - Record scope data using ziDAQServer's
%                                     synchronous poll function
% hf2_example_zsync_poll            - Synchronous demodulator sample timestamps
%                                     from multiple HF2s via the Zsync feature
% hf2_example_pid_advisor_pll       - Setup and optimize a PLL using the PID
%                                     Advisor
%
% EXAMPLES/UHF - Examples specific to the UHF Series
% uhf_example_awg                   - demonstrate different methods to
%                                     create waveforms and compile and
%                                     upload a SEQC program to the AWG
% uhf_example_awg_sourcefile        - demonstrate how to compile/upload a
%                                     SEQC from file.
% uhf_example_boxcar                - Record boxcar data using ziDAQServer's
%                                     synchronous poll function
%
% EXAMPLES/DEPRECATED - Examples that use functionality that either is or will
%                                     be made deprecated in a future release
```

% example_spectrum	- Perform an FFT using ziDAQ's zoomFFT module (Spectrum Tab of the LabOne UI)
%	
% example_swtrigger_edge	- Record demodulator data upon a rising edge trigger via ziDAQ's SW Trigger module
%	
% example_swtrigger_digital	- Record data using a digital trigger via ziDAQ's SW Trigger module
%	
% example_swtrigger_grid	- Record demodulator data, interpolated on a grid from multiple triggers using the SW Trigger's Grid Mode.
%	
% example_swtrigger_grid_average	- Record demodulator data, interpolated on a grid from multiple triggers using the SW Trigger's Grid Mode. This example additionally demonstrates Grid Mode's averaging functionality.
%	
%	
%	

Matlab Driver Naming

On Windows the MEX-file (the ziDAQ Matlab Driver/DLL) is called either `ziDAQ.mexw64` or `ziDAQ.mexw32` for 64-bit and 32-bit platforms respectively, on Linux it's called `ziDAQ.mexa64` and on Mac it's called `ziDAQ.mexmaci64`. When more than one MEX-file is present, Matlab automatically selects the correct MEX-file for the current platform.

4.2.2. Using the Built-in Documentation

To access `ziDAQ`'s documentation within Matlab, type either of the following in the Matlab Command Window:

```
>> help ziDAQ
```

```
>> doc ziDAQ
```

This documentation is located in the file `MATLAB2012/Driver/ziDAQ.m`. See [Section 4.5, LabOne Matlab API \(ziDAQ\) Command Reference](#) for a printer friendly version.

4.2.3. Running the Examples

Prerequisites for running the Matlab examples:

1. Matlab is configured for `ziDAQ` as described above in [Section 4.1](#).
2. The Data Server program is running and the instrument is discoverable, this is the case if the instrument can be seen in the User Interface.
3. Signal Output 1 of the instrument is connected to Signal Input 1 via a BNC cable; many of the Matlab examples measure on this hardware channel.

See [Section 4.2.1](#) for a list of available examples bundled with the LabOne Matlab API. All the examples follow the same structure and take one input argument: the device ID of the instrument they are to be ran with. For example:

```
>> example_sweeper('dev123');
```

The example should produce some output in the Matlab Command Window, such as:

```
ziDAQ version Jul 7 2015 accessing server localhost 8005.  
Will run the example on `dev123`, an `HF2LI` with options `MFK|PLL|MOD|RTK|PID`.  
Sweep progress 9%  
Sweep progress 19%  
Sweep progress 30%  
Sweep progress 42%  
Sweep progress 52%  
Sweep progress 58%  
Sweep progress 68%  
Sweep progress 79%  
Sweep progress 91%  
Sweep progress 100%  
ziDAQ: AtExit called
```

Most examples will also plot some data in a Matlab figure, see [Figure 4.1](#) for an example. If you encounter an error message please ensure that the [above prerequisites](#) are fulfilled and see [Section 4.4](#) for help troubleshooting the error.

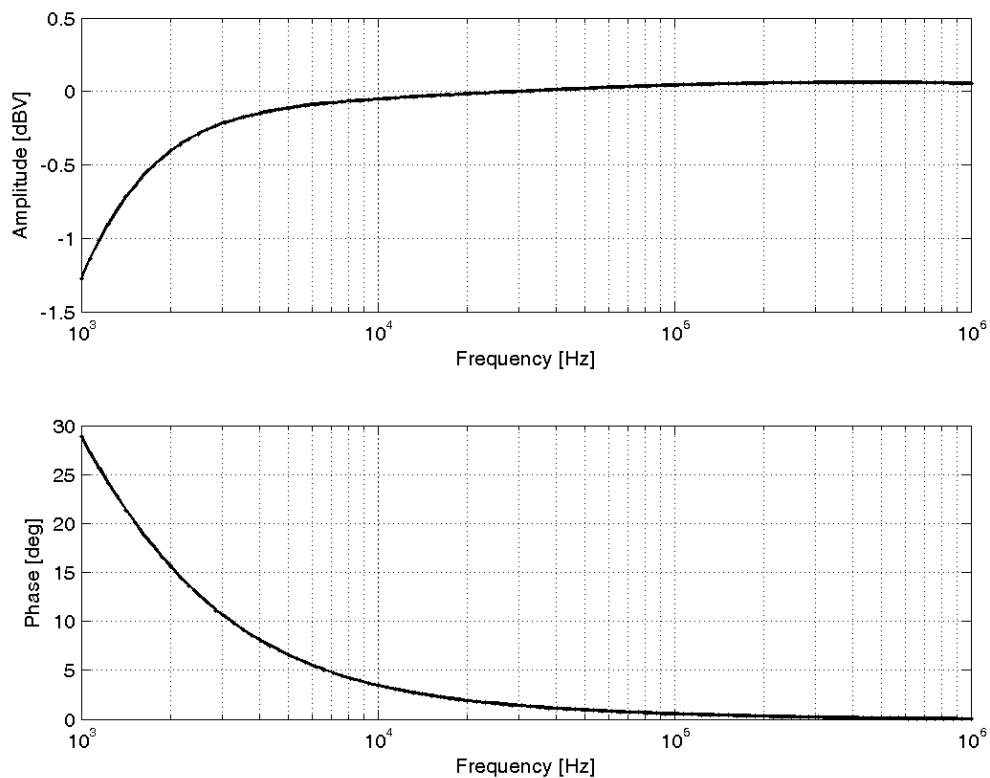


Figure 4.1. The plot produced by the LabOne Matlab API example `example_sweeper.m`; the plots show the instruments demodulator output when performing a frequency sweep over a simple feedback cable.

Note

The examples serve as a starting point for your own measurement needs. However, before editing the m-files, be sure to copy them to your own user space (they could be overwritten upon updating your LabOne installation) and give them a unique name to avoid name conflicts in Matlab.

4.2.4. Using ziCore Modules in the LabOne Matlab API

In the LabOne Matlab API [ziCore Modules](#) are configured and controlled via Matlab "handles". For example, in order to use the [Sweeper Module](#) a handle is created via:

```
>> h = ziDAQ('sweep');
```

and the Module's parameters are configured using the `set` command and specifying the Module's handle with a `path, value` pair, for example:

```
>> ziDAQ('set', h, 'sweep/start', 1.2e5);
```

The parameters can be read-back using the `get` command, which supports wildcards, for example:

```
>> sweep_params = ziDAQ('get', h, 'sweep/*');
```

The variable `sweep_params` now contains a struct of all the Sweeper's parameters. The other main Module commands are used similarly, e.g., `ziDAQ('execute', h)` to start the sweeper. See [Section 3.1.2](#) for more help with Modules and a description of their parameters.

Note

The Data Acquisition Module uses dot notation for subscribing to signals. In the data structure returned by the MATLAB API, the dots are replaced by underscores in order not to conflict with the dot notation used for member selection in MATLAB, e.g. `/devNNN/demods/0/sample.r` is accessed using `devNNN.demods(1).sample_r`.

4.2.5. Enabling Logging in the LabOne Matlab API

Logging from the API is not enabled by default upon initializing a server session with `ziDAQ`, it must be enabled (after using `connect`) with the `setDebugLevel` command. For example,

```
>> ziDAQ('setDebugLevel', 0);
```

sets the API's logging level to 0, which provides the most verbose logging output. The other log levels are defined as following:

```
trace:0, info:1, debug:2, warning:3, error:4, fatal:5, status:6.
```

It is also possible for the user to write their own messages directly to `ziDAQ`'s log using the `writeDebugLog` command. For example to write a log message of `info` severity level:

```
>> ziDAQ('writeDebugLog', 1, 'Hello log!');
```

On Windows the logs can be found by navigating to the Zurich Instruments "Logs" folder entry in the Windows Start Menu: Programs → Zurich Instruments → LabOne Servers → Logs. This will open an Explorer window displaying folders containing log files from various LabOne components, in particular, the `ziDAQLog` folder contains logs from the LabOne Matlab API. On Linux, the logs can be found at `"/tmp/ziDAQLog_USERNAME"`, where `"USERNAME"` is the same as the output of the `"whoami"` command.

4.3. LabOne Matlab API Tips and Tricks

In this section some tips and tricks for working with the LabOne Matlab API are provided.

The structure of **ziDAQ** commands.

All LabOne Matlab API commands are based on a call to the Matlab function `ziDAQ()`. The first argument to `ziDAQ()` specifies the API command to be executed and is an obligatory argument. For example, a session is instantiated between the API and the Data Server with the Matlab command `ziDAQ('connect')`. Depending on the type of command specified, optional arguments may be required. For example, to obtain an integer node value, the node path must be specified as a second argument to the `'getInt'` command:

```
s = ziDAQ('getInt','/dev123/sigouts/0/on');
```

where the output argument contains the current value of the specified node.

To set an integer node value, both the node path and the value to be set must be specified as the second and third arguments:

```
ziDAQ('setInt','/dev123/sigouts/0/on', 1);
```

See the [LabOne Matlab API \(ziDAQ\) Command Reference](#) for a list of all available commands.

Data Structures returned by **ziDAQ**.

The output arguments that `ziDAQ` returns are designed to use the native data structures that Matlab users are familiar with and that reflect the data's location in the instruments node hierarchy. For example, when the `poll` command returns data from the instruments fourth demodulator (located in the node hierarchy as `/dev123/demods/3/sample`), the output argument contains a nested `struct` in which the data can be accessed by

```
data = ziDAQ('poll', poll_length, poll_timeout);  
x = data.dev123.demods(4).sample.x;  
y = data.dev123.demods(4).sample.y;
```

The instrument's node tree uses zero-based indexing; Matlab uses one-based indexing.

See the tip [Data Structures returned by ziDAQ](#) : The **fourth** demodulator sample located at `/dev123/demods/3/sample`, is indexed in the data structure returned by `poll` as `data.dev123.demods(4).sample`.

Explicitly convert **uint64** data types to **double**.

Matlab's native data type is double-precision floating point and doesn't support performing calculations with other data types such as 64-bit unsigned integers, for example:

```
>> a = uint64(2); b = uint64(1); a - b  
? Undefined function or method 'minus' for input arguments of type 'uint64'.
```

Due to this limitation, be sure to convert demodulator timestamps to `double` before performing calculations. For example, in the following, both `clockbase` and `timestamp` (both 64-bit unsigned

integers) need to be converted to double before converting the timestamps from the instrument's native "ticks" to seconds via the instrument's clockbase:

```
data = ziDAQ('poll', 1.0, 500);           % poll data
sample = data.(device).demods(0).sample; % get the sample from the zeroth demod
% convert timestamps from ticks to seconds via the device's clockbase
% (the ADC's sampling rate), specify reference start time via t0.
clockbase = double(ziDAQ('getInt',['/' device '/clockbase']));
t = (double(sample.timestamp) - double(sample.timestamp(1)))/clockbase;
```

Use the utility function **ziCheckPathInData**.

Checking that a sub-structure in the nested data structure returned by `poll` actually exists can be cumbersome and can require multiple nested `if` statements; this can be avoided by using the utility function `ziCheckPathInData`. For example, the code:

```
data = ziDAQ('poll', poll_length, poll_timeout );
if isfield(data,device)
    if isfield(data.(device),'demods')
        if length(data.(device).demods) >= channel
            if ~isempty(data.(device).demods(channel).sample)
                % do something with the demodulator sample...
```

can be replaced by:

```
data = ziDAQ('poll', poll_length, poll_timeout );
if ziCheckPathInData( data, ['/' device '/demods/' demod_c '/sample']);
    % do something with the demodulator sample...
```

4.4. Troubleshooting the LabOne Matlab API

This section intends to solve possible error messages than can occur when using `ziDAQ` in Matlab.

Error message: "Undefined function or method 'ziDAQ' for input arguments of type '*'"

Matlab can not find the LabOne Matlab API library. Check whether the `MATLAB2012/Driver` subfolder of your LabOne installation is in the Matlab Search Path by using the command:

```
>> path
```

and repeating the steps to configure Matlab's search path in [Section 4.1.2](#).

Error message: "Undefined function or method 'example_sweeper'"

Matlab can not find the example. Check whether the `MATLAB2012/Examples/Common` subfolder (respectively `MATLAB2012/Examples/HDAWG`, `MATLAB2012/Examples/HF2` or `MATLAB2012/Examples/UHF`) of your LabOne installation are in the Matlab Search Path by using the command:

```
>> path
```

and repeating the steps to configure Matlab's search path in [Section 4.1.2](#).

Error message: "Error using: ziDAQ ZIAPIException with status code: 32870. Connection invalid."

The Matlab API can not connect to the Data Server. Please check that the correct port was used; that the correct server is running for your device and that the device is connected to the server, see [Section 1.4.1](#).

Error Message: "Error using: ziAutoConnect at 63 ziAutoConnect(): failed to find a running server or failed to find a connected a device..."

The utility function `ziAutoConnect()` located in `MATLAB2012/Utils/` tries to determine which Data Server is running and whether any devices are connected to that Data Server. It is only supported by UHFLI and HF2 Series instruments, MFLI instruments are not supported. Some suggestions to verify the problem:

- Please verify in the User Interface, whether a device is connected to the Data Server running on your computer.
- If the Data Server is running on a different computer, connect manually to the Data Server via `ziDAQ`'s `connect` function:

```
>> ziDAQ('connect', hostname, port, api_level);
```

where `hostname` should be replaced by the IP of the computer where the Data Server is running, `port` is specified as in [Section 1.4.1](#) and `api_level` is specified as described in [Section 1.4.2](#).

Error Message: "Error using: ziDAQ ZIAPIException on path /dev123/sigins/0/imp50 with status code: 16387. Value or Node not found"

The API is connected to the Data Server, but the command failed to find the specified node. Please:

- Check whether your instrument is connected to the Data Server in the User Interface; if it is not connected the instruments device node tree, e.g., /dev123/, will not be constructed by the Data Server.
- Check whether the node path is spelt correctly.
- Explore the node tree to verify the node actually exists with the `listNodes` command:

```
>> ziDAQ('listNodes', '/dev123/sigins/0', 3)
```

Error Message: "using: ziDAQ Server not connected. Use 'ziDAQ('connect', ...) first."

A `ziDAQ` command was issued before initializing a connection to the Data Server. First use the `connect` command:

```
>> ziDAQ('connect', hostname, port, api_level);
```

where `hostname` should be replaced by the IP address of the computer where the Data Server is running, `port` is specified as in [Section 1.4.1](#) and `api_level` is specified as described in [Section 1.4.2](#). If the Data Server is running on the same computer, use `'localhost'` as the `hostname`.

Error Message: "Attempt to execute SCRIPT ziDAQ as a function: ziDAQ.m"

There could be a problem with your LabOne Matlab API installation. The call to `ziDAQ()` is trying to call the help file `ziDAQ.m` as a function instead of calling the `ziDAQ()` function defined in the MEX-file. In this case you need to ensure that the `ziDAQ` MEX-file is in your search path as described in [Section 4.1](#) and navigate away from the `Driver` directory. Secondly, ensure that the LabOne Matlab MEX-file is in the `Driver` folder as described in [Section 4.2.1](#).

4.5. LabOne Matlab API (ziDAQ) Command Reference

```
%
% Copyright 2009-2018, Zurich Instruments Ltd, Switzerland
% This software is a preliminary version. Function calls and
% parameters may change without notice.
%
% This version of ziDAQ is linked against:
% * Matlab 7.9.0.529, R2009b, Windows,
% * Matlab 8.4.0.145, R2014b, Linux64.
% You can check which version of Matlab you are using Matlab's `ver` command.
% A list of compatible Matlab and ziDAQ versions is available here:
% www.zhinst.com/labone/compatibility
%
% ziDAQ is an interface for communication with Zurich Instruments Data Servers.
%
% Usage: ziDAQ(command, [option1], [option2])
%       command = 'awgModule', 'clear', 'connect', 'connectDevice',
%               'dataAcquisitionModule', 'deviceSettings',
%               'disconnectDevice', 'discoveryFind', 'discoveryGet',
%               'finished', 'flush', 'get', 'getAsEvent', 'getAuxInSample',
%               'getBytes', 'getComplex', 'getDIO', 'getDouble', 'getInt',
%               'getString', 'getSample', 'help', 'impedanceModule',
%               'listNodes', 'listNodesJSON', 'logOn',
%               'logOff', 'multiDeviceSyncModule', 'pidAdvisor',
%               'poll', 'pollEvent', 'programRT', 'progress',
%               'read', 'record', 'setByte', 'setComplex', 'setDouble',
%               'syncSetDouble', 'setInt', 'syncSetInt', 'setString',
%               'syncSetString', 'subscribe', 'sweep', 'unsubscribe',
%               'update', 'zoomFFT'
%
% Preconditions: ZI Server must be running (check task manager)
%
%       ziDAQ('connect', [host = '127.0.0.1'], [port = 8005], [apiLevel = 1]);
%       [host] = Server host string (default is localhost)
%       [port] = Port number (double)
%               Use port 8005 to connect to the HF2 Data Server
%               Use port 8004 to connect to the MF or UHF Data Server
%       [apiLevel] = Compatibility mode of the API interface (int64)
%               Use API level 1 to use code written for HF2.
%               Higher API levels are currently only supported
%               for MF and UHF devices. To get full functionality for
%               MF and UHF devices use API level 5.
%       To disconnect use 'clear ziDAQ'
%
%       result = ziDAQ('getConnectionAPILevel');
%       Returns ziAPI level used for the active connection.
%
%       result = ziDAQ('discoveryFind', device);
%       device (string) = Device address string (e.g. 'UHF-DEV2000')
%       Return the device ID for a given device address.
%
%       result = ziDAQ('discoveryGet', deviceId);
%       deviceId (string) = Device id string (e.g. DEV2000)
%       Return the device properties for a given device id.
%
%       ziDAQ('connectDevice', device, interface);
%       device (string) = Device serial to connect (e.g. 'DEV2000')
%       interface (string) = Interface, e.g., 'USB', '1GbE'.
%       Connect with the data server to a specified device over the
%       specified interface. The device must be visible to the server.
%       If the device is already connected the call will be ignored.
%       The function will block until the device is connected and
```

```

%           the device is ready to use. This method is useful for UHF
%           devices offering several communication interfaces.
%
%           ziDAQ('disconnectDevice', device);
%           device (string) = Device serial of device to disconnect.
%           This function will return immediately. The disconnection of
%           the device may not yet finished.
%
% result = ziDAQ('listNodes', path, flags);
%           path (string) = Node path or partial path, e.g.,
%                           '/dev100/demods/'.
%           flags (int64) = Define which nodes should be returned, set the
%                           following bits to obtain the described behavior:
%                           int64(0) -> ZI_LIST_NODES_NONE 0x00
%                               The default flag, returning a simple
%                               listing of the given node
%                           int64(1) -> ZI_LIST_NODES_RECURSIVE 0x01
%                               Returns the nodes recursively
%                           int64(2) -> ZI_LIST_NODES_ABSOLUTE 0x02
%                               Returns absolute paths
%                           int64(4) -> ZI_LIST_NODES_LEAFONLY 0x04
%                               Returns only nodes that are leafs,
%                               which means the they are at the
%                               outermost level of the tree.
%                           int64(8) -> ZI_LIST_NODES_SETTINGSONLY 0x08
%                               Returns only nodes which are marked
%                               as setting
%                           int64(16) -> ZI_LIST_NODES_STREAMINGONLY 0x10
%                               Returns only streaming nodes
%                           int64(32) -> ZI_LIST_NODES_SUBSCRIBEDONLY 0x20
%                               Returns only subscribed nodes
%           Returns a list of nodes with description found at the specified
%           path. Flags may also be combined, e.g., set flags to bitor(1, 2)
%           to return paths recursively and printed as absolute paths.
%
% result = ziDAQ('listNodesJSON', path, flags);
%           path (string) = Node path or partial path, e.g.,
%                           '/dev100/demods/'.
%           flags (int64) = Define which nodes should be returned, set the
%                           following bits to obtain the described behavior.
%                           They are the same as for listNodes(), except that
%                           0x01, 0x02 and 0x04 are enforced:
%                           int64(8) -> ZI_LIST_NODES_SETTINGSONLY 0x08
%                               Returns only nodes which are marked
%                               as setting
%                           int64(16) -> ZI_LIST_NODES_STREAMINGONLY 0x10
%                               Returns only streaming nodes
%                           int64(32) -> ZI_LIST_NODES_SUBSCRIBEDONLY 0x20
%                               Returns only subscribed nodes
%                           int64(64) -> ZI_LIST_NODES_BASECHANNEL 0x40
%                               Return only one instance of a node in case of multiple
%                               channels
%                           int64(64) -> ZI_LIST_NODES_GETONLY 0x80
%                               Return only nodes which can be used with the get
%                               command
%           Returns a list of nodes with description found at the specified
%           path as a JSON formatted string. Only UHF and MF devices support
%           this functionality. Flags may also be combined, e.g., set flags
%           to bitor(1, 2) to return paths recursively and printed as
%           absolute paths.
%
% result = ziDAQ('help', path);
%           path (string) = Node path or partial path, e.g.,
%                           '/dev100/demods/'.
%           Returns a formatted description of the nodes in the supplied path.
%           Only UHF and MF devices support this functionality.
%

```

```
% result = ziDAQ('getSample', path);
%     path (string) = Node path
%     Returns a single demodulator sample (including
%     DIO and AuxIn). For more efficient data recording
%     use the subscribe and poll functions.
%
% result = ziDAQ('getAuxInSample', path);
%     path (string) = Node path
%     Returns a single auxin sample. Note, the auxin data
%     is averaged in contrast to the auxin data embedded
%     in the demodulator sample.
%
% result = ziDAQ('getDIO', path);
%     path (string) = Node path.
%     Returns a single DIO sample.
%
% result = ziDAQ('getDouble', path);
%     path (string) = Node path
%
% result = ziDAQ('getComplex', path);
%     path (string) = Node path
%
% result = ziDAQ('getInt', path);
%     path (string) = Node path
%
% result = ziDAQ('getBytes', path);
%     path (string) = Node path
%
% result = ziDAQ('getString', path);
%     path (string) = Node path
%
%     ziDAQ('setDouble', path, value);
%     path (string) = Node path
%     value (double) = Setting value
%
%     ziDAQ('setComplex', path, value);
%     path (string) = Node path
%     value (complex double) = Setting value
%
%     ziDAQ('syncSetDouble', path, value);
%     Deprecated, see the 'sync' command.
%     path (string) = Node path
%     value (double) = Setting value
%
%     ziDAQ('setInt', path, value);
%     path (string) = Node path
%     value (int64) = Setting value
%
%     ziDAQ('syncSetInt', path, value);
%     Deprecated, see the 'sync' command.
%     path (string) = Node path
%     value (int64) = Setting value
%
%     ziDAQ('setByte', path, value);
%     path (string) = Node path
%     value (double) = Setting value
%
%     ziDAQ('setString', path, value);
%     path (string) = Node path
%     value (string) = Setting value
%
%     ziDAQ('syncSetString', path, value);
%     path (string) = Node path
%     value (string) = Setting value
%
%     ziDAQ('asyncSetString', path, value);
%     path (string) = Node path
```

```

%           value (string) = Setting value
%
%   ziDAQ('vectorWrite', path, value);
%       path (string) = Vector node path
%       value (vector of (u)int8, (u)int16, (u)int32, (u)int64,
%           float, double; or string) = Setting value
%
%   ziDAQ('subscribe', path);
%       path (string) = Node path
%       Subscribe to the specified path to receive streaming data
%       or setting data if changed. Use either 'poll' command to
%       obtain the subscribed data.
%
%   ziDAQ('unsubscribe', path);
%       path (string) = Node path
%       Unsubscribe from the node paths specified via 'subscribe'.
%       Use a wildcard ('*') to unsubscribe from all data.
%
%   ziDAQ('getAsEvent', path);
%       path (string) = Node path
%       Triggers a single event on the path to return the current
%       value. The result can be fetched with the 'poll' or 'pollEvent'
%       command.
%
%   ziDAQ('update');
%       Detect HF2 devices connected to the USB. On Windows this
%       update is performed automatically.
%
%   ziDAQ('get', path, [settingsOnly]);
%       path (string) = Node path
%       Gets a structure of the node data from the specified
%       branch. High-speed streaming nodes (e.g. /devN/demods/0/sample)
%       are not returned. Wildcards (*) may be used, in which case
%       read-only nodes are ignored.
%       [settingsOnly] (uint32) = Specify which type of nodes to include
%       in the result. Allowed:
%           ZI_LIST_NODES_SETTINGSONLY = 8 (default)
%           ZI_LIST_NODES_NONE = 0 (all nodes)
%
%   ziDAQ('flush');
%       Deprecated, see the 'sync' command.
%       Flush all data in the socket connection and API buffers.
%       Call this function before a subscribe with subsequent poll
%       to get rid of old streaming data that might still be in
%       the buffers.
%
%   ziDAQ('echoDevice', device);
%       Deprecated, see the 'sync' command.
%       device (string) = device serial, e.g. 'dev100'.
%       Sends an echo command to a device and blocks until
%       answer is received. This is useful to flush all
%       buffers between API and device to enforce that
%       further code is only executed after the device executed
%       a previous command.
%
%   ziDAQ('sync');
%       Synchronize all data paths. Ensures that get and poll
%       commands return data which was recorded after the
%       setting changes in front of the sync command. This
%       sync command replaces the functionality of all 'syncSet*',
%       'flush', and 'echoDevice' commands.
%
%   ziDAQ('programRT', device, filename);
%       device (string) = device serial, e.g. 'dev100'.
%       filename (string) = filename of RT program.
%       HF2 devices only; writes down a real-time program. Requires
%       the Real time Option must be available for the specified

```

```

%           HF2 device.
%
%   result = ziDAQ('secondsTimeStamp', [timestamps]);
%           timestamps (uint64) = vector of uint64 device ticks
%           Deprecated. In order to convert timestamps to seconds divide the
%           timestamps by the value of the instrument's clockbase device node,
%           e.g., /dev99/clockbase.
%           [Converts a timestamp vector of uint64 ticks
%           into a double vector of timestamps in seconds (HF2 Series).]
%
% Synchronous Interface
%
%   ziDAQ('poll', duration, timeout, [flags]);
%           duration (double) = Time in [s] to continuously check for value
%           changes in subscribed nodes before
%           returning
%           timeout (int64) = Poll timeout in [ms]; recommended: 10 ms
%           [flags] (uint32) = Flags specifying data polling properties
%           Bit[0] FILL : Fill data loss holes
%           Bit[2] THROW : Throw if data loss is detected (only
%           possible in combination with FILL).
%           Bit[3] DETECT: Just detect data loss holes.
%           Continuously check for value changes (by calling pollEvent) in
%           all subscribed nodes for the specified duration and return the
%           data. If no value change occurs in subscribed nodes before
%           duration + timeout, poll returns no data. This function call is
%           blocking (it is synchronous). However, since all value changes
%           are returned since either subscribing to the node or the last
%           poll (assuming no buffer overflow has occurred on the Data
%           Server), this function may be used in a quasi-asynchronous
%           manner to return data spanning a much longer time than the
%           specified duration. The timeout parameter is only relevant when
%           communicating in a slow network. In this case it may be set to
%           a value larger than the expected round-trip time in the
%           network.
%
%   result = ziDAQ('pollEvent', timeout);
%           timeout (int64) = Poll timeout in [ms]
%           Return the value changes that occurred in one single subscribed
%           node. This is a low-level function. The poll function is better
%           suited in nearly all cases.
%
%% LabOne API Modules
% Shared Interface (common for all modules)
%
%   result = ziDAQ('listNodes', handle, path, flags);
%           handle = Matlab handle (reference) specifying an instance of
%           the module class.
%           path (string) = Module parameter path
%           flags (int64) = Define which module parameters paths should be
%           returned, set the following bits to obtain the
%           described behaviour:
%           flags = int64(0) -> ZI_LIST_NODES_NONE 0x00
%           The default flag, returning a simple
%           listing of the given path
%           int64(1) -> ZI_LIST_NODES_RECURSIVE 0x01
%           Returns the paths recursively
%           int64(2) -> ZI_LIST_NODES_ABSOLUTE 0x02
%           Returns absolute paths
%           int64(4) -> ZI_LIST_NODES_LEAFSONLY 0x04
%           Returns only paths that are leafs,
%           which means the they are at the
%           outermost level of the tree.
%           int64(8) -> ZI_LIST_NODES_SETTINGSONLY 0x08
%           Returns only paths which are marked
%           as setting
%           Flags may also be combined, e.g., set flags to bitor(1, 2)

```



```
%          to return paths recursively and printed as absolute paths.
%
%      ziDAQ('subscribe', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Node path to process data received from the
%                device. Use wildcard ('*') to select all.
%          Subscribe to device nodes. Call multiple times to
%          subscribe to multiple node paths. After subscription the
%          processing can be started with the 'execute'
%          command. During the processing paths can not be
%          subscribed or unsubscribed.
%
%      ziDAQ('unsubscribe', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Node path to process data received from the
%                device. Use wildcard ('*') to select all.
%          Unsubscribe from one or several nodes. During the
%          processing paths can not be subscribed or
%          unsubscribed.
%
%      ziDAQ('getInt', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Path string of the module parameter. Must
%                start with the module name.
%          Get integer module parameter. Wildcards are not supported.
%
%      ziDAQ('getDouble', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Path string of the module parameter. Must
%                start with the module name.
%          Get floating point double module parameter. Wildcards are not
%          supported.
%
%      ziDAQ('getString', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Path string of the module parameter. Must
%                start with the module name.
%          Get string module parameter. Wildcards are not supported.
%
%      ziDAQ('get', handle, path);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Path string of the module parameter. Must
%                start with the module name.
%          Get module parameters. Wildcards are supported, e.g. 'sweep/*'.
%
%      ziDAQ('set', handle, path, value);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          path (string) = Path string of the module parameter. Must
%                start with the module name.
%          value = The value to set the module parameter to, see the list
%                of module parameters for the correct type.
%          Set the specified module parameter value.
%
%      ziDAQ('execute', handle);
%          handle = Matlab handle (reference) specifying an instance of
%                the module class.
%          Start the module thread. Subscription or unsubscription
%          is not possible until the module is finished.
%
%      result = ziDAQ('finished', handle);
```

```

%             handle = Matlab handle (reference) specifying an instance of
%             the module class.
%             Returns 1 if the processing is finished, otherwise 0.
%
% result = ziDAQ('read', handle);
%             handle = Matlab handle (reference) specifying an instance of
%             the module class.
%             Read out the recorded data; transfer the module data to
%             Matlab.
%
%             ziDAQ('finish', handle);
%             handle = Matlab handle (reference) specifying an instance of
%             the module class.
%             Stop executing. The thread may be restarted by
%             calling 'execute' again.
%
% result = ziDAQ('progress', handle);
%             handle = Matlab handle (reference) specifying an instance of
%             the module class.
%             Report the progress of the measurement with a number
%             between 0 and 1.
%
%             ziDAQ('clear', handle);
%             handle = Matlab handle (reference) specifying an instance of
%             the module class.
%             Stop the module's thread.
%
%% Sweep Module
%
% handle = ziDAQ('sweep', timeout);
%             timeout = Poll timeout in [ms] - DEPRECATED, ignored
%             Creates a sweep class. The thread is not yet started.
%             Before the thread start subscribe and set command have
%             to be called. To start the real measurement use the
%             execute function.
%
% Sweep Parameters
% sweep/device          string  Device that should be used for
%                               the parameter sweep, e.g. 'dev99'.
% sweep/start          double  Sweep start frequency [Hz]
% sweep/stop           double  Sweep stop frequency [Hz]
% sweep/gridnode       string  Path of the node that should be
%                               used for sweeping. For frequency
%                               sweep applications this will be e.g.
%                               'oscs/0/freq'. The device name of
%                               the path can be omitted and is given
%                               by sweep/device.
% sweep/loopcount      int     Number of sweep loops (default 1)
% sweep/endless        int     Endless sweeping (default 0)
%                               0 = Use loopcount value
%                               1 = Endless sweeping enabled, ignore
%                               loopcount
% sweep/samplecount    int     Number of samples per sweep
% sweep/settling/time   double  Settling time before measurement is
%                               performed, in [s]
% sweep/settling/tc     double  Settling precision
%                               5 ~ low precision
%                               15 ~ medium precision
%                               50 ~ high precision
% sweep/settling/inaccuracy
%                               int  Demodulator filter settling inaccuracy
%                               that defines the wait time between a
%                               sweep parameter change and recording of
%                               the next sweep point. The settling time
%                               is calculated as the time required to
%                               attain the specified remaining proportion

```

```

% [1e-13, 0.1] of an incoming step
% function. Typical inaccuracy
% values:
% - 10m for highest sweep speed for large
% signals,
% - 100u for precise amplitude measurements,
% - 100n for precise noise measurements.
% Depending on the order the settling
% accuracy will define the number of filter
% time constants the sweeper has to
% wait. The maximum between this value and
% the settling time is taken as wait time
% until the next sweep point is recorded.
%
% sweep/xmapping          int    Sweep mode
%                               0 = linear
%                               1 = logarithmic
%
% sweep/scan              int    Scan type
%                               0 = sequential
%                               1 = binary
%                               2 = bidirectional
%                               3 = reverse
%
% sweep/bandwidth          double Fixed bandwidth [Hz]
%                               0 = Automatic calculation (obsolete)
%
% sweep/bandwidthcontrol  int    Sets the bandwidth control mode (default 2)
%                               0 = Manual (user sets bandwidth and order)
%                               1 = Fixed (uses fixed bandwidth value)
%                               2 = Auto (calculates best bandwidth value)
%                               Equivalent to the obsolete bandwidth = 0
%                               setting
%
% sweep/bandwidthoverlap  bool   Sets the bandwidth overlap mode (default 0). If
%                               enabled the bandwidth of a sweep point may
%                               overlap with the frequency of neighboring sweep
%                               points. The effective bandwidth is only limited
%                               by the maximal bandwidth setting and omega
%                               suppression. As a result, the bandwidth is
%                               independent of the number of sweep points. For
%                               frequency response analysis bandwidth overlap
%                               should be enabled to achieve maximal sweep
%                               speed (default: 0).
%                               0 = Disable
%                               1 = Enable
%
% sweep/order              int    Defines the filter roll off to use in Fixed
%                               bandwidth selection.
%                               Valid values are between 1 (6 dB/octave)
%                               and 8 (48 dB/octave). An order of 0
%                               triggers a read-out of the order from the
%                               selected demodulator.
%
% sweep/maxbandwidth       double Maximal bandwidth used in auto bandwidth
%                               mode in [Hz]. The default is 1.25MHz.
%
% sweep/omegasuppression  double Damping in [dB] of omega and 2omega components.
%                               Default is 40dB in favor of sweep speed.
%                               Use higher value for strong offset values or
%                               3omega measurement methods.
%
% sweep/averaging/tc       double Min averaging time [tc]
%                               0 = no averaging (see also time!)
%                               5 ~ low precision
%                               15 ~ medium precision
%                               50 ~ high precision
%
% sweep/averaging/sample  int    Min samples to average
%                               1 = no averaging (if averaging/tc = 0)
%
% sweep/averaging/time     double Min averaging time [s]
%
% sweep/phaseunwrap        bool   Enable unwrapping of slowly changing phase
%                               evolutions around the +/-180 degree boundary.
%
% sweep/sincfilter         bool   Enables the sinc filter if the sweep frequency
%                               is below 50 Hz. This will improve the sweep
%                               speed at low frequencies as omega components
%                               do not need to be suppressed by the normal

```

```

%
% sweep/awgcontrol      bool    low pass filter.
%                               Enable AWG control for sweeper. If enabled
%                               the sweeper will automatically start the AWG
%                               and records the sweep sample based on the
%                               even index in hwtrigger.
% sweep/save/directory  string  The base directory where files are saved.
% sweep/save/filename   string  Defines the sub-directory where files are
%                               saved. The actual sub-directory have this
%                               name with a sequence count appended, e.g.
%                               sweep_000.
% sweep/save/fileformat string  The format of the file for saving data:
%                               0 = Matlab,
%                               1 = CSV,
%                               2 = ZView (Impedance data only).
% sweep/save/csvseparator
%                               string The character to use as CSV separator when
%                               saving files in this format.
% sweep/save/csvlocale  string  The locale to use for the decimal point character
%                               and digit grouping character for numerical values
%                               in CSV files:
%                               "C" (default)      = dot for the decimal point and
%                               no digit grouping,
%                               "" (empty string) = use the symbols set in the
%                               language and region settings
%                               of the computer.
% sweep/save/save       bool    Initiate the saving of data to file. The saving
%                               is done in the background When the save is
%                               finished, this parameter goes low.
% sweep/save/saveonread bool    Automatically save the data to file
%                               immediately before reading the data out of
%                               read() command. Set this parameter when
%                               you want to save data to file when running
%                               the module in continuous mode while doing
%                               intermediate reads.
% sweep/historylength   bool    Maximum number of entries stored in the
%                               measurement history.
% sweep/clearhistory    bool    Remove all records from the history list.
%
% Note:
% Settling time = max(settling.tc * tc, settling.time)
% Averaging time = max(averaging.tc * tc, averaging.sample / sample-rate)
%
%% Device Settings Module
%
% handle = ziDAQ('deviceSettings', timeout);
%         timeout = Poll timeout in [ms] - DEPRECATED, ignored
%         Creates a device settings class for saving/loading device
%         settings to/from a file. Before starting the module, set the path,
%         filename and command parameters. To run the command, use the
%         execute function.
%
% Device Settings Parameters
% deviceSettings/device      string  Comma separated list of devices that
%                                   should be used for loading/saving
%                                   device settings, e.g. 'dev99,dev100'.
% deviceSettings/path        string  Path where the settings files are to
%                                   be located. If not set, the default
%                                   settings location of the LabOne
%                                   software is used.
% deviceSettings/filename    string  The file to which the settings are to
%                                   be saved/loaded.
% deviceSettings/command     string  The save/load command to execute.
%                                   'save' = Read device settings and save
%                                   to file.
%                                   'load' = Load settings from file and
%                                   write to device.

```

```

%                                     'read' = Read device settings only
%                                     (no save).
%
%
%% PLL Advisor Module, DEPRECATED (use PID Advisor instead)
%
%   The PLL advisor module for the UHF is removed and fully replaced
%   by the generic PID advisor module for all Zurich Instruments devices.
%
%% PID Advisor Module
%
%   PID Advisor Parameters
%   pidAdvisor/advancedmode    int    Disable automatic calculation of the
%                                       start and stop value.
%   pidAdvisor/auto            int    Automatic response calculation triggered
%                                       by parameter change.
%   pidAdvisor/bode            struct  Output parameter. Contains the resulting
%                                       bode plot of the PID simulation.
%   pidAdvisor/bw              double  Output parameter. Calculated system
%                                       bandwidth.
%   pidAdvisor/calculate       int    In/Out parameter. Command to calculate
%                                       values. Set to 1 to start the
%                                       calculation.
%   pidAdvisor/display/freqstart double Start frequency for Bode plot.
%                                       For disabled advanced mode the start
%                                       value is automatically derived from the
%                                       system properties.
%   pidAdvisor/display/freqstop double Stop frequency for Bode plot.
%   pidAdvisor/display/timestart double Start time for step response.
%   pidAdvisor/display/timestop double Stop time for step response.
%   pidAdvisor/dut/bw          double  Bandwidth of the DUT (device under test).
%   pidAdvisor/dut/damping     double  Damping of the second order
%                                       low pass filter.
%   pidAdvisor/dut/delay       double  IO Delay of the feedback system
%                                       describing the earliest response for
%                                       a step change.
%   pidAdvisor/dut/fcenter     double  Resonant frequency of the of the modelled
%                                       resonator.
%   pidAdvisor/dut/gain        double  Gain of the DUT transfer function.
%   pidAdvisor/dut/q           double  quality factor of the modelled resonator.
%   pidAdvisor/dut/source      int    Type of model used for the external
%                                       device to be controlled by the PID.
%                                       source = 1: Low-pass first order
%                                       source = 2: Low-pass second order
%                                       source = 3: Resonator frequency
%                                       source = 4: Internal PLL
%                                       source = 5: VCO
%                                       source = 6: Resonator amplitude
%   pidAdvisor/impulse         struct  Output parameter. Impulse response
%                                       (not yet supported).
%   pidAdvisor/index           int    PID index for parameter detection.
%   pidAdvisor/pid/autobw      int    Adjusts the demodulator bandwidth to fit
%                                       best to the specified target bandwidth
%                                       of the full system.
%   pidAdvisor/pid/d           double  In/Out parameter. Differential gain.
%   pidAdvisor/pid/dlimittimeconstant double In/Out parameter. Differential filter
%                                       timeconstant.
%   pidAdvisor/pid/i           double  In/Out parameter. Integral gain.
%   pidAdvisor/pid/mode        double  Select PID Advisor mode. Mode value is
%                                       bit coded, bit 0: P, bit 1: I, bit 2: D,
%                                       bit 3: D filter limit.
%   pidAdvisor/pid/p           double  In/Out parameter. Proportional gain.
%   pidAdvisor/pid/rate        double  In/Out parameter. PID Advisor sampling
%                                       rate of the PID control loop.
%   pidAdvisor/pid/targetbw    double  PID system target bandwidth.
%   pidAdvisor/todevice        int    Set to 1 to transfer PID advisor

```

```

% data to the device.
% pidAdvisor/type          string  If set to 'pll' for an HF2 device the
%                               PID advisor will advise parameters for
%                               the hardware PLL. The default is 'pid'.
%                               This parameter is only relevant for HF2
%                               devices.
% pidAdvisor/pm            double  Output parameter. Simulated phase margin
%                               of the PID with the current settings.
%                               The phase margin should be greater than
%                               45 deg and preferably greater than 65 deg
%                               for stable conditions.
% pidAdvisor/pmfreq        double  Output parameter. Simulated phase margin
%                               frequency.
% pidAdvisor/stable        int     Output parameter. When 1, the PID Advisor
%                               found a stable solution with the given
%                               settings. When 0, revise your settings
%                               and rerun the PID Advisor.
% pidAdvisor/step          struct  Output parameter. Contains the resulting
%                               step response plot of the PID simulation.
% pidAdvisor/targetbw      double  Requested PID bandwidth. Higher
%                               frequencies may need manual tuning.
% pidAdvisor/targetfail    int     Output parameter. 1 indicates the
%                               simulated PID BW is smaller than the
%                               Target BW.
% pidAdvisor/tf/closedloop int     Switch the response calculation mode
%                               between closed or open loop.
% pidAdvisor/tf/input      int     Start point for the plant response
%                               simulation for open or closed loops.
% pidAdvisor/tf/output     int     End point for the plant response
%                               simulation for open or closed loops.
% pidAdvisor/tune          int     Optimize the PID parameters so that
%                               the noise of the closed-loop
%                               system gets minimized. The HF2 doesn't
%                               support tuning.
% pidAdvisor/tuner/mode     int     Select tuner mode. Mode value
%                               is bit coded, bit 0: P, bit 1: I,
%                               bit 2: D, bit 3: D filter limit.
% pidAdvisor/tuner/averagetime double Time for a tuner iteration.
%
% handle = ziDAQ('pidAdvisor', timeout);
%         timeout = Poll timeout in [ms] - DEPRECATED, ignored
%         Creates a PID Advisor class for simulating the PID in the
%         device. Before the thread start, set the command parameters,
%         call execute() and then set the "calculate" parameter to start
%         the simulation.
%
%% AWG Module
%
% handle = ziDAQ('awgModule');
%         Creates an AWG compiler class for compiling the AWG sequence and
%         pattern downloaded to the device .
%
% AWG Module Parameters
%
% Path name          Type      Description
% awgModule/compiler/sourcefile  string  AWG sequencer program file to load.
%                                     The file needs to be saved in the 'src'
%                                     sub-directory of the AWG settings
%                                     directory.
% awgModule/compiler/sourcestring string  AWG sequencer program string to load.
%                                     Allows to compile a sequencer program
%                                     without saving it to a file first.
%                                     Compilation will start directly after
%                                     setting of the parameter. Setting
%                                     awgModule/compiler/start is not
%                                     necessary.

```

%	awgModule/compiler/waveforms	string	Comma-separated list waveform files to be used by the AWG sequencer program.
%	awgModule/compiler/statusstring	string	Status message of the compiler (read only)
%	awgModule/compiler/status	int	Status of the compiler (read only):
%			-1 = idle
%			0 = compilation successful
%			1 = compilation failed
%			2 = compilation encountered warnings
%	awgModule/compiler/start	bool	Start compilation using the source file specified by
%			awgModule/compiler/sourcefile
%			and upload of the AWG sequencer program to the device. Will be reset after completion or on error.
%	awgModule/device	string	Device that should be used to run AWG sequencer programs, e.g. 'dev99'
%	awgModule/directory	string	Directory where AWG sequencer programs, waveforms and ELF files should be located. If not set, the default settings location of the LabOne software is used.
%	awgModule/elf/file	string	File name of the ELF file to upload. If not set, the name will be set automatically based on the source file name. The file will be saved in the 'elf' sub-directory of the AWG settings directory.
%	awgModule/elf/upload	bool	Start upload of the AWG sequencer program to the device. Will be reset after completion or on error.
%	awgModule/elf/status	int	Status of the ELF file upload (read-only).
%			-1 = idle
%			0 = upload successful
%			1 = upload failed
%			2 = upload is in progress
%	awgModule/elf/checksum	int	Checksum of the uploaded ELF file (read-only).
%	awgModule/progress	double	Reports the progress of the upload with a number between 0 and 1.
%			
%	%% Impedance Module		
%			
%	handle = ziDAQ('impedanceModule');		
%	Creates a impedance class for executing a user compenastion.		
%			
%	Impedance Module Parameters		
%			
%	Path name	Type	Description
%	impedanceModule/directory	string	The directory where files are saved.
%	impedanceModule/calibrate	bool	If set to true will execute a compensation for the specified compensation condition.
%	impedanceModule/device	string	Device string defining the device on which the compensation is performed.
%	impedanceModule/step	int	Compensation step to be performed when calibrate indicator is set to true.
%			0 - First load
%			1 - Second load
%			2 - Third load
%			3 - Fourth load
%	impedanceModule/mode	int	Compensation mode to be used. Defines which load steps need to be compensated.

%			3 - SO (Short-Open)
%			4 - L (Load)
%			5 - SL (Short-Load)
%			6 - OL (Open-Load)
%			7 - SOL (Short-Open-Load)
%			8 - LLL (Load-Load-Load)
%	impedanceModule/status	int	Bit coded field of the already compensated load conditions (bit 0 = first load).
%	impedanceModule/loads/0/r	double	Resistance value of first compensation load (SHORT) [Ohm]
%	impedanceModule/loads/1/r	double	Resistance value of second compensation load (OPEN) [Ohm]
%	impedanceModule/loads/2/r	double	Resistance value of third compensation load (LOAD) [Ohm]
%	impedanceModule/loads/3/r	double	Resistance value of the fourth compensation load (LOAD) [Ohm]. This load setting is only used if high impedance load is enabled.
%	impedanceModule/loads/0/c	double	Parallel capacitance of the first compensation load (SHORT) [F]
%	impedanceModule/loads/1/c	double	Parallel capacitance of the second compensation load (OPEN) [F]
%	impedanceModule/loads/2/c	double	Parallel capacitance of the third compensation load (LOAD) [F]
%	impedanceModule/loads/3/c	double	Parallel capacitance of the fourth compensation load (LOAD) [F]
%	impedanceModule/freq/start	double	Start frequency of compensation traces [Hz]
%	impedanceModule/freq/stop	double	Stop frequency of compensation traces [Hz]
%	impedanceModule/freq/samplecount	int	Number of samples of a compensation trace
%	impedanceModule/highimpedanceload	bool	Enable a second high impedance load compensation for the low current ranges.
%	impedanceModule/expectedstatus	int	Bit field of the load condition that the corresponds a full compensation. If status is equal the expected status the compensation is complete.
%	impedanceModule/message	string	Message string containing information, warnings or error messages during compensation.
%	impedanceModule/comment	string	Comment string that will be saved together with the compensation data.
%	impedanceModule/validation	bool	Enable the validation of compensation data. If enabled the compensation is checked for too big deviation from specified load.
%	impedanceModule/precision	int	Precision of the compensation. Will affect time of a compensation and reduces the noise on compensation traces.
%			0 - Standard speed
%			1 - Low speed / high precision
%	impedanceModule/todevice	bool	If enabled will automatically transfer compensation data to the persistent flash memory in case of a valid compensation.
%	impedanceModule/progress	double	Progress of a compensation condition.
%	%% Scope Module		
%	Scope Module Parameters		
%	Path name	Type	Description

%	scopeModule/historylength	int	Maximum number of entries stored in the measurement history.
%	scopeModule/clearhistory	bool	Remove all records from the history list.
%	scopeModule/externalscaling	double	Scaling to apply to the scope data transferred over API level 1 connection (HF2).
%	scopeModule/mode	int	Scope data processing mode:
%			0 - Pass through: scope segments assembled and returned unprocessed, non-interleaved.
%			1 - Moving average: entire scope recording assembled, scaling applied, averager if enabled (see weight), data returned in float non-interleaved format.
%			2 - Average: not implemented yet.
%			3 - FFT: same as mode 1, except FFT applied to every segment of the scope recording. See fft/* nodes for FFT parameters.
%	scopeModule/averager/weight	int	=1 - averager disabled. >1 - moving average, updating last history entry.
%	scopeModule/averager/restart	bool	1 - resets the averager. Action node, switches back to 0 automatically.
%	scopeModule/averager/resamplingmode	int	When averaging low sampling rate data aligned by high resolution trigger, scope data must be resampled to keep same samples positioning relative to the trigger between averaged recordings. Resample using:
%			0 - Linear interpolation
%			1 - PCHIP interpolation
%	scopeModule/fft/window	int	FFT Window:
%			0 - Rectangular
%			1 - Hann
%			2 - Hamming
%			3 - Blackman Harris 4 term
%			16 - Exponential (ring-down)
%			17 - Cosine (ring-down)
%			18 - Cosine squared (ring-down)
%	scopeModule/fft/power	bool	1 - Calculate power value
%	scopeModule/fft/spectraldensity	bool	1 - Calculate spectral density value
%	scopeModule/save/directory	string	The base directory where files are saved.
%	scopeModule/save/filename	string	Defines the sub-directory where files are saved. The actual sub-directory have this name with a sequence count appended, e.g. zoomFFT_000.
%	scopeModule/save/fileformat	string	The format of the file for saving data:
%			0 = Matlab,
%			1 = CSV,
%			2 = ZView (Impedance data only).
%	scopeModule/save/csvseparator	string	The character to use as CSV separator when saving files in this format.
%	scopeModule/save/csvlocale	string	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files:
%			"C" (default) = dot for the decimal point and no digit grouping,
%			"" (empty string) = use the symbols set in the language and region settings of the computer.
%	scopeModule/save/save	bool	Initiate the saving of data to file. The

```

%
%
% scopeModule/save/saveonread bool saving is done in the background When the
% save is finished, this parameter goes low.
% Automatically save the data to file
% immediately before reading the data out of
% read() command. Set this parameter when
% you want to save data to file when running
% the module in continuous mode while doing
% intermediate reads.
%% Multi-Device Sync Module
%
% Multi-Device Sync Module Parameters
% Path name Type Description
% multiDeviceSyncModule/devices string Defines which instruments should be
% included in the synchronization. Expects
% a comma-separated list of devices in the
% order the devices are connected.
% multiDeviceSyncModule/group int Defines in which synchronization group
% should be accessed by the module.
% multiDeviceSyncModule/message int Status message of the module.
% multiDeviceSyncModule/start bool Set to true to start the synchronization
% process.
% multiDeviceSyncModule/status int Status of the synchronization process.
% -1 - error
% 0 - idle
% 1 - synchronization in progress
% 2 - synchronization successful
% multiDeviceSyncModule/recover bool Set to enable recovery of already
% synchronized devices.
% 0 - off (default)
% 1 - on
% multiDeviceSyncModule/phasesync bool Set to 1 to start phase reset on"\n"
% alloscillators of synches"\n"
% devices.")
%
%
% Data Acquisition Module
%
% handle = ziDAQ('dataAcquisitionModule', handle);
% handle = Matlab handle (reference) specifying an instance of
% the DataAcquisitionModule class.
% Create an instance of the Data Acquisition Module class
% and return a Matlab handle with which to access it.
% Before the thread can actually be started (via 'execute'):
% - the desired data to record must be specified via the module's
% 'subscribe' command,
% - the device serial (e.g., dev100) that will be used must be
% set.
% The real measurement is started upon calling the 'execute'
% function. After that the module will start recording data and
% verifying for incoming triggers.
% Force a trigger to manually record one duration of the
% subscribed data.
%
%
% Data Acquisition Module Parameters
% dataAcquisitionModule/buffercount int The number of buffers used for
% data acquisition (read-only).
% dataAcquisitionModule/buffersize double The buffersize [s] of the Data
% Acquisition Module object (read-only).
% dataAcquisitionModule/flags int Record flags.
% FILL = 0x0001 : Not supported by module.
% Grid mode will always
% fill data loss holes.
% ALIGN = 0x0002 : Not supported by module.

```

```

%                                     Grid Mode will always
%                                     align data.
%                                     THROW = 0x0004 : Throw if sample loss is
%                                     detected.
%                                     DETECT = 0x0008: Just detect data loss
%                                     holes. This flag is
%                                     always enabled.
% dataAcquisitionModule/device string The device serial to use
%                                     the Data Acquisition Module with, e.g.
%                                     dev123 (compulsory
%                                     parameter).
% dataAcquisitionModule/enable bool Enable the module.
% dataAcquisitionModule/endless bool Enable endless triggering
%                                     1=enable; 0=disable.
% dataAcquisitionModule/fft/absolute
%                                     bool Shifts the frequencies so that the center
%                                     frequency becomes the demodulation
%                                     frequency rather than 0 Hz.
% dataAcquisitionModule/fft/window
%                                     int FFT window (default 1 = Hann)
%                                     0 = Rectangular
%                                     1 = Hann
%                                     2 = Hamming
%                                     3 = Blackman Harris 4 term
%                                     16 - Exponential (ring-down)
%                                     17 - Cosine (ring-down)
%                                     18 - Cosine squared (ring-down)
% dataAcquisitionModule/forcetrigger bool Force a trigger.
% dataAcquisitionModule/triggered
%                                     bool Indicates a trigger event.
% dataAcquisitionModule/awgcontrol
%                                     bool Enable interaction with AWG. If enabled
%                                     the hwtrigger index counter will be used
%                                     to control the grid row for recording.
% dataAcquisitionModule/triggernode
%                                     string Path and signal of the node that should be
%                                     used for triggering, separated by a
%                                     dot (.), e.g. /devN/demods/0/sample.x
% dataAcquisitionModule/count int Number of trigger edges to record.
% dataAcquisitionModule/type int Trigger type used. Some parameters are
%                                     only valid for special trigger types.
%                                     0 = trigger off
%                                     1 = analog edge trigger on source
%                                     2 = digital trigger mode on DIO source
%                                     3 = analog pulse trigger on source
%                                     4 = analog tracking trigger on source
%                                     5 = change trigger
%                                     6 = hardware trigger on trigger line
%                                     source
%                                     7 = tracking edge trigger on source
%                                     8 = event count trigger on counter source
% dataAcquisitionModule/edge int Trigger edge
%                                     1 = rising edge
%                                     2 = falling edge
%                                     3 = both
% dataAcquisitionModule/findlevel
%                                     bool Automatically find the value of
%                                     dataAcquisitionModule/level based on the
%                                     current signal value.
% dataAcquisitionModule/bits int Digital trigger condition.
% dataAcquisitionModule/bitmask
%                                     int Bit masking for bits used for triggering.
%                                     Used for digital trigger.
% dataAcquisitionModule/delay double Trigger frame position [s] (left side)
%                                     relative to trigger edge.
%                                     delay = 0 -> trigger edge at left border.
%

```

```

%                                     delay < 0 -> trigger edge inside trigger
%                                     frame (pretrigger).
%                                     delay > 0 -> trigger edge before trigger
%                                     frame (posttrigger).
% dataAcquisitionModule/duration
%                                     double Data acquisition frame length [s]
% dataAcquisitionModule/level         double Trigger level voltage [V].
% dataAcquisitionModule/hysteresis
%                                     double Trigger hysteresis [V].
% dataAcquisitionModule/triggered
%                                     bool    Has the Module triggered?
%                                     1=Yes, 0=No (read only).
% dataAcquisitionModule/bandwidth
%                                     double Filter bandwidth [Hz] for pulse and
%                                     tracking triggers.
% dataAcquisitionModule/holdoff/count
%                                     int     Number of skipped triggers until the next
%                                     trigger is recorded again.
% dataAcquisitionModule/holdoff/time
%                                     double Hold off time [s] before the next trigger
%                                     is acquired again. A hold off time smaller
%                                     than the duration will produce overlapped
%                                     trigger frames.
% dataAcquisitionModule/pulse/min
%                                     double Minimum pulse width [s] for the pulse
%                                     trigger.
% dataAcquisitionModule/pulse/max
%                                     double Maximum pulse width [s] for the pulse
%                                     trigger.
% dataAcquisitionModule/eventcount/mode
%                                     int     Specifies the mode used for event count
%                                     processing.
%                                     0 - Trigger on every event count sample
%                                     1 - Trigger if event count value
%                                     incremented
% dataAcquisitionModule/grid/mode
%                                     int     Specify how the captured data is mapped
%                                     onto the grid. Each trigger becomes a row
%                                     in the matrix and each trigger's data is
%                                     mapped onto a new grid row defined by the
%                                     number of columns using this setting:
%                                     1: Use nearest neighbour interpolation.
%                                     2: Use with linear interpolation.
%                                     4: Use exact alignment to the grid.
%                                     In this mode the duration is
%                                     determined from the number of grid
%                                     columns and the highest data sampling
%                                     rate of the signals to be captured.
% dataAcquisitionModule/grid/overwrite
%                                     int     If enabled, the module will return only
%                                     one data chunk when it is running, which
%                                     is overwritten by consecutive triggers.
% dataAcquisitionModule/grid/cols
%                                     int     Specify the number of columns in the
%                                     grid's matrix. The data from each row is
%                                     mapped onto the grid according to the
%                                     grid/mode setting with the
%                                     specified number of columns.
% dataAcquisitionModule/grid/rows
%                                     int     Specify the number of rows in the grid's
%                                     matrix. Each row is the data recorded from
%                                     one trigger mapped onto the columns.
% dataAcquisitionModule/grid/repetitions
%                                     int     Number of statistical operations performed
%                                     per grid.
% dataAcquisitionModule/grid/direction
%                                     int     The direction to organize data in the

```

```
% grid's matrix:
0: Forward. The data in each row is ordered chronologically, e.g., the first data point in each row corresponds to the first timestamp in the trigger data.
1: Reverse. The data in each row is ordered reverse chronologically, e.g., the first data point in each row corresponds to the last timestamp in the trigger data.
2: Bidirectional. The ordering of the data alternates between Forward and Backward ordering from row-to-row. The first row is Forward ordered.

dataAcquisitionModule/grid/waterfall
    int If enabled, the last grid row is always updated and the previously captured rows are shifted by one.

dataAcquisitionModule/refreshrate
    double Set the rate at which the triggers are processed (Hz).

dataAcquisitionModule/save/directory
    string The base directory where files are saved.

dataAcquisitionModule/save/filename
    string Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g.swttrigger_000.

dataAcquisitionModule/save/fileformat
    string The format of the file for saving data:
        0 = Matlab,
        1 = CSV,
        2 = SXM (Image format).

dataAcquisitionModule/save/csvseparator
    string The character to use as CSV separator when saving files in this format.

dataAcquisitionModule/save/csvlocale
    string The locale to use for the decimal point character and digit grouping character for numerical values in CSV files:
        "C" (default)      = dot for the decimal point and no digit grouping,
        "" (empty string) = use the symbols set in the language and region settings of the computer.

dataAcquisitionModule/save/save
    bool Initiate the saving of data to file. The saving is done in the background. When the save is finished, this parameter goes low.

dataAcquisitionModule/save/saveonread
    bool Automatically save the data to file immediately before reading the data out of read() command. Set this parameter when you want to save data to file when running the module in continuous mode while doing intermediate reads.

dataAcquisitionModule/historylength
    bool Sets an upper limit for the number of data captures stored in the module.

dataAcquisitionModule/preview bool When enabled, allows the data of an incomplete trigger to be read. Useful for long data acquisitions/FFTs to display the progress.
```

```

% dataAcquisitionModule/clearhistory
%                               bool    Clear all captured data from the module.
%
% dataAcquisitionModule/spectrum/autobandwidth
%                               bool    When set to 1, initiates automatic
%                                       adjustment of the demodulator bandwidths
%                                       to obtain optimal alias rejection for the
%                                       selected frequency span which is
%                                       equivalent to the sampling rate.
%                                       The FFT mode has to be enabled
%                                       (spectrum/enable) and the module has
%                                       to be running for this function
%                                       to take effect.
%
% dataAcquisitionModule/spectrum/enable
%                               bool    Enables the FFT mode of the data
%                                       Acquisition module, in addition to
%                                       time domain.
%
% dataAcquisitionModule/spectrum/frequencyspan
%                               double  Sets the desired frequency span of
%                                       the FFT.
%
% dataAcquisitionModule/spectrum/overlapped
%                               bool    Enables overlapping FFTs. If disabled (0),
%                                       FFTs are performed on distinct abutting
%                                       data sets. If enabled, the data sets of
%                                       successive FFTs overlap based on the
%                                       defined refresh rate.
%
%% Debugging Functions
%
% ziDAQ('setDebugLevel', debuglevel);
% debuglevel (int) = Debug level (trace:0, info:1, debug:2,
% warning:3, error:4, fatal:5, status:6).
% Enables debug log and sets the debug level.
%
% ziDAQ('writeDebugLog', severity, message);
% severity (int) = Severity (trace:0, info:1, debug:2, warning:3,
% error:4, fatal:5, status:6).
% message (str) = Message to output to the log.
% Outputs message to the debug log (if enabled).
%
% ziDAQ('logOn', flags, filename, [style]);
% flags = LOG_NONE:           0x00000000
%         LOG_SET_DOUBLE:     0x00000001
%         LOG_SET_INT:        0x00000002
%         LOG_SET_BYTE:       0x00000004
%         LOG_SET_STRING:     0x00000008
%         LOG_SYNC_SET_DOUBLE: 0x00000010
%         LOG_SYNC_SET_INT:    0x00000020
%         LOG_SYNC_SET_BYTE:   0x00000040
%         LOG_SYNC_SET_STRING: 0x00000080
%         LOG_GET_DOUBLE:      0x00000100
%         LOG_GET_INT:         0x00000200
%         LOG_GET_BYTE:        0x00000400
%         LOG_GET_STRING:      0x00000800
%         LOG_GET_DEMOD:       0x00001000
%         LOG_GET_DIO:         0x00002000
%         LOG_GET_AUXIN:       0x00004000
%         LOG_GET_COMPLEX:     0x00008000
%         LOG_LISTNODES:       0x00010000
%         LOG_SUBSCRIBE:       0x00020000
%         LOG_UNSUBSCRIBE:     0x00040000
%         LOG_GET_AS_EVENT:    0x00080000
%         LOG_UPDATE:          0x00100000

```

```

%             LOG_POLL_EVENT:      0x00200000
%             LOG_POLL:            0x00400000
%             LOG_ALL :            0xffffffff
%
% filename = Log file name
% [style] = LOG_STYLE_TELNET: 0 (default)
%             LOG_STYLE_MATLAB: 1
%             LOG_STYLE_PYTHON: 2
%
% Log all API commands sent to the Data Server. This is useful
% for debugging.
%
%
% ziDAQ('logOff');
%     Turn of message logging.
%
%
% %% SW Trigger Module (this module will be made deprecated in a future release - new
% users should use the DAQ Module instead).
%
%
% handle = ziDAQ('record' duration, timeout);
%
%     duration (double) = The module's internal buffersize to use when
% recording data [s]. The recommended size is
% 2*trigger/0/duration parameter. Note that
% this can be modified via the
% trigger/buffersize parameter.
% DEPRECATED, set 'buffersize' param instead.
%
%     timeout (int64) = Poll timeout [ms]. - DEPRECATED, ignored
% Create an instance of the ziDAQRecorder and return a Matlab
% handle with which to access it.
% Before the thread can actually be started (via 'execute'):
% - the desired data to record must be specified via the module's
%   'subscribe' command,
% - the device serial (e.g., dev100) that will be used must be
%   set.
% The real measurement is started upon calling the 'execute'
% function. After that the trigger will start recording data and
% verifying for incoming triggers.
%
%
% ziDAQ('trigger', handle);
%
%     handle = Matlab handle (reference) specifying an instance of
% the ziDAQRecorder class.
% Force a trigger to manually record one duration of the
% subscribed data.
%
%
% Trigger Parameters
%
%     trigger/buffersize      double Set the buffersize [s] of the trigger
% object. The recommended buffer size is
% 2*trigger/0/duration.
%
%     trigger/flags           int    Record flags.
%                                     FILL = 0x0001 : Fill holes.
%                                     ALIGN = 0x0002 : Align data that contains a
%                                     timestamp.
%                                     THROW = 0x0004 : Throw if sample loss
%                                     is detected.
%                                     DETECT = 0x0008: Just detect data loss holes.
%
%     trigger/device          string The device serial to use the software trigger
% with, e.g. dev123 (compulsory parameter).
%
%     trigger/endless         bool   Enable endless triggering 1=enable;
% 0=disable.
%
%     trigger/forcetrigger    bool   Force a trigger.
%
%     trigger/awgcontrol      bool   Enable interaction with AWG. If enabled the
% hwtrigger index counter will be used to
% control the grid row for recording.
%
%     trigger/0/triggernode   string Path and signal of the node that should be
% used for triggering, separated by a dot (.),
% e.g. /devN/demods/0/sample.x
% Overrides values from trigger/0/path and
% trigger/0/source.
%
%     trigger/0/path          string The path to the demod sample to trigger on,

```

%			e.g. demods/3/sample, see also
%			trigger/0/source
%			DEPRECATED - use trigger/0/triggernode
%			instead
%	trigger/0/source	int	Signal that is used to trigger on.
%			0 = x
%			1 = y
%			2 = r
%			3 = angle
%			4 = frequency
%			5 = phase
%			6 = auxiliary input 0 / parameter 0
%			7 = auxiliary input 1 / parameter 1
%			DEPRECATED - use trigger/0/triggernode
%			instead
%	trigger/0/count	int	Number of trigger edges to record.
%	trigger/0/type	int	Trigger type used. Some parameters are
%			only valid for special trigger types.
%			0 = trigger off
%			1 = analog edge trigger on source
%			2 = digital trigger mode on DIO source
%			3 = analog pulse trigger on source
%			4 = analog tracking trigger on source
%			5 = change trigger
%			6 = hardware trigger on trigger line
%			source
%			7 = tracking edge trigger on source
%			8 = event count trigger on counter source
%	trigger/0/edge	int	Trigger edge
%			1 = rising edge
%			2 = falling edge
%			3 = both
%	trigger/0/findlevel	bool	Automatically find the value of
%			trigger/0/level
%			based on the current signal value.
%	trigger/0/bits	int	Digital trigger condition.
%	trigger/0/bitmask	int	Bit masking for bits used for
%			triggering. Used for digital trigger.
%	trigger/0/delay	double	Trigger frame position [s] (left side)
%			relative to trigger edge.
%			delay = 0 -> trigger edge at left border.
%			delay < 0 -> trigger edge inside trigger
%			frame (pretrigger).
%			delay > 0 -> trigger edge before trigger
%			frame (posttrigger).
%	trigger/0/duration	double	Recording frame length [s]
%	trigger/0/level	double	Trigger level voltage [V].
%	trigger/0/hysteresis	double	Trigger hysteresis [V].
%	trigger/0/retrigger	int	Record more than one trigger in a trigger
%			frame. If a trigger event is currently being
%			recorded and another trigger event is
%			detected within the duration of the current
%			trigger event, extend the size of the
%			trigger frame to include the duration of the
%			new trigger event.
%	trigger/triggered	bool	Has the software trigger triggered? 1=Yes,
%			0=No (read only).
%	trigger/0/bandwidth	double	Filter bandwidth [Hz] for pulse and
%			tracking triggers.
%	trigger/0/holdoff/count	int	Number of skipped triggers until the
%			next trigger is recorded again.
%	trigger/0/holdoff/time	double	Hold off time [s] before the next
%			trigger is recorded again. A hold off
%			time smaller than the duration will
%			produce overlapped trigger frames.
%	trigger/0/hwtrigsources	int	Only available for devices that support
%			hardware triggering. Specify the channel


```

%                                     to trigger on.
%                                     DEPRECATED - use trigger/0/triggernode
%                                     instead
% trigger/0/pulse/min                double Minimal pulse width [s] for the pulse
%                                     trigger.
% trigger/0/pulse/max                double Maximal pulse width [s] for the pulse
%                                     trigger.
% trigger/0/eventcount/mode          int   Specifies the mode used for event count
%                                     processing.
%                                     0 - Trigger on every event count sample
%                                     1 - Trigger if event count value incremented
% trigger/0/grid/mode                int   Enable grid mode. In grid mode a matrix
%                                     instead of a vector is returned. Each
%                                     trigger becomes a row in the matrix and each
%                                     trigger's data is interpolated onto a new
%                                     grid defined by the number of columns:
%                                     0: Disable
%                                     1: Enable with nearest neighbour
%                                     interpolation
%                                     2: Enable with linear interpolation.
% trigger/0/grid/operation            int   If running in endless mode, either replace
%                                     or average the data in the grid's matrix.
% trigger/0/grid/cols                int   Specify the number of columns in the grid's
%                                     matrix. The data from each row is
%                                     interpolated onto a grid with the specified
%                                     number of columns.
% trigger/0/grid/rows                int   Specify the number of rows in the grid's
%                                     matrix. Each row is the data recorded from
%                                     one trigger interpolated onto the columns.
% trigger/0/grid/repetitions          int   Number of statistical operations performed
%                                     per grid.
% trigger/0/grid/direction            int   The direction to organize data in the grid's
%                                     matrix:
%                                     0: Forward.
%                                     The data in each row is ordered chrono-
%                                     logically, e.g., the first data point in
%                                     each row corresponds to the first
%                                     timestamp in the trigger data.
%                                     1: Reverse.
%                                     The data in each row is ordered reverse
%                                     chronologically, e.g., the first data
%                                     point in each row corresponds to the last
%                                     timestamp in the trigger data.
%                                     2: Bidirectional.
%                                     The ordering of the data alternates
%                                     between Forward and Backward ordering from
%                                     row-to-row. The first row is Forward
%                                     ordered.
% trigger/save/directory              string The base directory where files are saved.
% trigger/save/filename               string Defines the sub-directory where files
%                                     are saved. The actual sub-directory
%                                     have this name with a sequence count
%                                     appended, e.g. swtrigger_000.
% trigger/save/fileformat              string The format of the file for saving data:
%                                     0 = Matlab,
%                                     1 = CSV,
%                                     2 = ZView (Impedance data only).
% trigger/save/csvseparator            string The character to use as CSV separator when
%                                     saving files in this format.
% trigger/save/csvlocale              string The locale to use for the decimal point
%                                     character and digit grouping character
%                                     for numerical values in CSV files:
%                                     "C" (default) = dot for the decimal
%                                     point and no digit
%                                     grouping,
%                                     "" (empty string) = use the symbols
%                                     set in the language

```

```

%                                     and region settings
%                                     of the computer.
% trigger/save/save          bool    Initiate the saving of data to file.
%                               The saving is done in the background
%                               When the save is finished, this
%                               parameter goes low.
% trigger/save/saveonread    bool    Automatically save the data to file
%                                     immediately before reading the data out of
%                                     read() command. Set this parameter when
%                                     you want to save data to file when running
%                                     the module in continuous mode while doing
%                                     intermediate reads.
% trigger/historylength      bool    Maximum number of entries stored in the
%                                     measurement history.
% trigger/clearhistory       bool    Remove all records from the history list.
%
%% Spectrum Module (this module will be made deprecated in a future release - new
% users should use the DAQ Module instead).
%
% handle = ziDAQ('zoomFFT', timeout);
%             timeout = Poll timeout in [ms] - DEPRECATED, ignored
%             Creates a zoom FFT class. The thread is not yet started.
%             Before the thread start subscribe and set command have
%             to be called. To start the real measurement use the
%             execute function.
%
% Zoom FFT Parameters (brief description, see the Programming Manual for more
% information)
% zoomFFT/device             string   Device that should be used for
%                                     the FFT, e.g. 'dev99'.
% zoomFFT/bit                int      Number of FFT points 2^bit
% zoomFFT/mode               int      Signal source for the FFT.
%                                     0 = Perform FFT on X+iY
%                                     1 = Perform FFT on R
%                                     2 = Perform FFT on Phase
%                                     3 = Perform FFT on Frequency
%                                     4 = Perform FFT on Phase derivative
% zoomFFT/loopcount          int      Number of zoom FFT loops (default 1)
% zoomFFT/endless            int      Run the frequency analysis continuously
%                                     (default 0)
%                                     0 = endless off, use loopcount value
%                                     1 = endless on, ignore loopcount
% zoomFFT/overlap            double   Overlap of the demod data used
%                                     for the FFT, 0 = none, [0..1]
% zoomFFT/settling/time      double   Settling time before measurement is performed
% zoomFFT/settling/tc        double   Settling time in time constant units before
%                                     the FFT recording is started.
%                                     5 ~ low precision
%                                     15 ~ medium precision
%                                     50 ~ high precision
% zoomFFT/window             int      FFT window (default 1 = Hann)
%                                     0 = Rectangular
%                                     1 = Hann
%                                     2 = Hamming
%                                     3 = Blackman Harris 4 term
%                                     16 - Exponential (ring-down)
%                                     17 - Cosine (ring-down)
%                                     18 - Cosine squared (ring-down)
% zoomFFT/absolute           bool     Shifts the frequencies so that the center
%                                     frequency becomes the demodulation frequency
%                                     rather than 0 Hz.
% zoomFFT/save/directory     string   The base directory where files are saved.
% zoomFFT/save/filename      string   Defines the sub-directory where files are
%                                     saved. The actual sub-directory have this
%                                     name with a sequence count appended, e.g.

```

```

% zoomFFT_000.
% zoomFFT/save/fileformat
% string The format of the file for saving data:
% 0 = Matlab,
% 1 = CSV,
% 2 = ZView (Impedance data only).
% zoomFFT/save/csvseparator
% string The character to use as CSV separator when
% saving files in this format.
% zoomFFT/save/csvlocale
% string The locale to use for the decimal point
% character and digit grouping character for
% numerical values in CSV files:
% "C" (default) = dot for the decimal
% point and no digit
% grouping,
% "" (empty string) = use the symbols set in
% the language and region
% settings of the
% computer.
% zoomFFT/save/save bool Initiate the saving of data to file. The
% saving is done in the background When the
% save is finished, this parameter goes low.
% zoomFFT/save/saveonread
% bool Automatically save the data to file
% immediately before reading the data out of
% read() command. Set this parameter when
% you want to save data to file when running
% the module in continuous mode while doing
% intermediate reads.
%

```

Chapter 5. Python Programming

The Zurich Instruments LabOne Python API is distributed as the [zhinst](#) Python package via PyPi, the official third-party software repository for Python. The `zhinst` package contains the `ziPython` binary extension that is used to communicate with Zurich Instruments data servers and devices. It allows users to configure and stream data from their instrument directly into a Python programming environment using Python's own native data structures and `numpy` arrays.

This chapter aims to help you get started using the Zurich Instruments LabOne Python API to control your instrument, please refer to:

- [Section 5.1 for help Installing the LabOne Python API.](#)
- [Section 5.2 for help Getting Started with the LabOne Python API and Running the Examples.](#)
- [Section 5.3 for LabOne Python API Tips and Tricks.](#)
- [Section 5.4 for the LabOne Python API \(ziPython\) Command Reference.](#)

About Python

Python is open source software, freely available for download from [Python's official website](#). Python is a high-level programming language with an extensive standard library renowned for its "batteries included" approach. Combined with the [numpy](#) package for scientific computing, Python is a powerful computational tool for scientists that does not require an expensive software license.

This chapter and the provided examples are not intended to be a Python tutorial. For help getting started with Python itself, see either the [Python Tutorial](#) or one of the many online resources, for example, the [learnpython.org](#). The [Interactive Python Course](#) is an interesting resource for those already familiar with Python basics.

5.1. Installing the LabOne Python API

This section lists detailed requirements. In most cases, installing the LabOne Python API should be as simple as searching for and installing the `zhinst` package in your Python distribution's package manager or running the command-line command:

```
pip install zhinst
```

5.1.1. Requirements

The following requirements must be fulfilled in order to install and use the LabOne Python API:

1. One of the following supported platforms and Python versions:
 - a. 32 or 64-bit Windows with a Python 2.7, 3.5, 3.6 or 3.7 installation.
 - b. 64-bit Linux with a Python 2.7, 3.5, 3.6 or 3.7 installation.
 - c. 64-bit Mac OS X and the system Python 2.7 (shipped pre-installed with OS X; other Python installations are not supported). The Data Server, which is unavailable on OS X, must run remotely on either an instrument or on a separate Windows or Linux PC.
2. The `pip` Python package, Python's most popular package management system.
3. The `numpy` Python package (automatically installed by `pip` if `zhinst` is installed over the internet).

Although the newest version of `numpy` is recommended, the `zhinst` package supports the following `numpy` versions:

Python 2.7: `>=1.6`

Python 3.5: `>=1.10`

Python 3.6: `>=1.12`

Python 3.7: `>=1.14`

4. The correct version of the `zhinst` Python package for the target Python version and platform (automatically detected by `pip` if `zhinst` is installed over the internet).

Note, it's necessary to first uninstall/remove `zhinst` when down or upgrading to release 14.08 and it's recommend to first uninstall `zhinst` when down or upgrading to release 18.12. See below for more details.

Upgrading to the LabOne Python API release 14.08

Important: If you your system already has an existing `ziPython` installation older than version 14.08, please be sure to either manually uninstall `ziPython` or manually remove the existing `zhinst` installation folder. This is due to improvements in the `zhinst` package structure in 14.08 (examples for different device classes are now organized in separate module/sub-directories) and the Python installer simply overwrites the existing installation, leading to a duplication of some files. For help locating `[PYTHONROOT]\lib\site-packages\zhinst\` on your system, please see [the section called “Locating the zhinst Installation Folder and Examples”](#).

Packaging changes to the LabOne Python API in 18.12

The LabOne Python API prior to LabOne software release 18.12 was distributed as an MSI installer (Windows) or tarball (Linux, Mac), downloadable from the Zurich Instruments [download page](#). From LabOne software release 18.12 onwards the LabOne Python API is distributed as a

Python wheel which can be either downloaded via pip (or manually) from PyPi as the `zhinst` Python package. If installing `zhinst` from a wheel for the first time it is recommended to uninstall the `ziPython` installer first in Windows. On Linux, if `zhinst` was installed using pip, it's recommended to first uninstall the old installation before installing the wheel:

```
pip uninstall ziPython
```

Note, in releases prior to 18.12, the Python API package was called `ziPython`. Since the top-level package is actually called `zhinst` and `ziPython` is the binary extension contained within it, the Python API package was renamed in 18.12 to `zhinst` for consistency.

5.1.2. Recommended Python Packages

The following Python packages can additionally be useful for programming with the LabOne Python API:

1. `matplotlib` - recommended to plot the output from many of `zhinst`'s examples.
2. `scipy` - recommended to load data saved from the LabOne UI in binary Matlab format (.mat).

5.1.3. Installation (Windows, Linux, Mac OS X)

The following installs the `zhinst` package from PyPi over the internet locally for the user performing the installation and does not require administrator rights. If the target PC for installation does not have access to the internet, please additionally see [Offline Installation](#).

1. Determine the path to the target Python installation. If the Python executable is not in your path, you can obtain the full path to your Python executable as follows:

```
from __future__ import print_function
import sys
print(sys.executable)
```

On Windows this will print something similar to:

```
C:\Python27\python.exe
```

2. Install the `zhinst` package. Using the full path to the Python executable, `PATH_TO_PYTHON_EXE`, as determined above in Step 1, open a command prompt and run Python with the `pip` module to install the `zhinst` package:

```
PATH_TO_PYTHON_EXE -m pip install --user zhinst
```

The `--user` flag tells `pip` to install the `zhinst` package locally for the user executing the command. Normally administrator rights are required in order to install the `zhinst` package for all users of the computer, for more information see below.

Global Installation as Administrator

In order to install the `zhinst` package for all users of the target Python installation, it must be installed using administrator rights and `pip`'s `--user` command-line flag should not be used. On Windows, Step 2 must be ran in a command prompt opened with administrator rights, this is normally achieved by doing a mouse right-click on the shortcut to `cmd.exe` and selecting "Run as administrator". On Linux, the package can be installed by preceding the installation step by "sudo".

5.1.4. Offline Installation

To install `zhinst` package on a computer without access to the internet, please download the correct wheel file for your system and Python version from <https://pypi.org/project/zhinst/> from another computer and copy it to the offline computer. If the target Python installation is Python 2.7 on a Linux 64-bit computer, please see [Determining the correct Unicode version for Python 2.7 distributions](#) below. If the `numpy` package is not yet installed, it can also be downloaded from either <https://pypi.org/project/numpy/> or from Christoph Gohlke's [pythonlibs page](#). Then the wheels can be installed as described above using `pip`, except that the name of the wheel file must be provided as the last argument to `pip` instead of the name of the package, `zhinst`.

Determining the correct Unicode version for Python 2.7 distributions

In order to determine which version of Unicode your Python 2.7 distribution uses, please type the following commands in the interactive shell of your target Python distribution:

```
import sys
print sys.maxunicode
```

If the last command prints:

- 65535, use the UCS2 version of `zhinst`:

```
zhinst-xx.xx.xxxxx-cp27-cp27m-manylinux1_x86_64.whl
```

- 1114111, use the UCS4 version of `zhinst` (note the additional "u"):

```
zhinst-xx.xx.xxxxx-cp27-cp27mu-manylinux1_x86_64.whl
```

5.2. Getting Started with the LabOne Python API

This section introduces the user to the LabOne Python API.

5.2.1. Contents of the LabOne Python API

Alongside the binary extension `ziPython` for interfacing with Zurich Instruments Data Servers and devices, the LabOne Python API includes utility functions and examples. See:

- [Section 5.4.1](#) to see which examples are available in `zhinst`.
- [Section 5.4.2](#) to see which utility functions are available in `zhinst`.

5.2.2. Using the Built-in Documentation

`ziPython`'s built-in documentation can be accessed using the `help` command in a python interactive shell:

- On module level:

```
>>> import zhinst.ziPython as ziPython
>>> help(ziPython)
```

- On class level, for example, for the Sweeper Module:

```
>>> import zhinst.ziPython as ziPython
>>> help(ziPython.SweeperModule)
```

- On function level, for example, for the `ziDAQServer.poll` method:

```
>>> import zhinst.ziPython as ziPython
>>> help(ziPython.ziDAQServer.poll)
```

See [Section 5.4, LabOne Python API \(ziPython\) Command Reference](#) for a printer friendly version of the built-in documentation.

5.2.3. Running the Examples

Prerequisites for running the Python examples:

1. The `zhinst` package is installed as described above in [Section 5.1](#).
2. The Data Server program is running and the instrument is discoverable, this is the case if the instrument can be seen in the User Interface.
3. Signal Output 1 of the instrument is connected to Signal Input 1 via a BNC cable; many of the Python examples measure on this hardware channel.

It's also recommended to install the `matplotlib` Python package in order to plot the data obtained in many of the examples, see [Section 5.1.2](#).

The API examples are available in the module `zhinst.examples`, which is organized into sub-modules according to the target Instrument class:

- `zhinst.examples.common`: examples compatible with any class of instrument,
- `zhinst.examples.hdawg`: examples only compatible with HDAWG Instruments,
- `zhinst.examples.hf2`: examples only compatible with HF2 Series Instruments,
- `zhinst.examples.uhf`: examples only compatible with the UHF Lock-in Amplifier.

All the examples follow the same structure and take one input argument: The device ID of the instrument to run the example with. The recommended way to run a `zhinst` example is to import

the example's module in an interactive shell and call the `run_example()` function. For example, to run the [Data Acquisition Module FFT](#) example:

```
import zhinst.examples.common.example_data_acquisition_edge_fft as example
example.run_example('dev123', do_plot=True);
```

which will produce some output in the Python shell, such as:

```
Discovered device `dev196`: HF2LI with options MFK, PLL, MOD, RTK, PID, WEB.
Creating an API session for device `dev196` on `127.0.0.1`, `8005` with apilevel `1`.
Setting trigger/0/level to 0.221.
Setting trigger/0/hysteresis 0.011.
Data Acquisition Module progress (acquiring 10 triggers): 100.00%.
Trigger is finished.
Data Acquisition's read() returned 10 signal segments.
```

Specify `do_plot=False` if matplotlib is unavailable. If matplotlib is installed, most examples will also plot the retrieved data, see [Figure 5.1](#) for an example. If you encounter an error message please ensure that the [above prerequisites](#) are fulfilled.

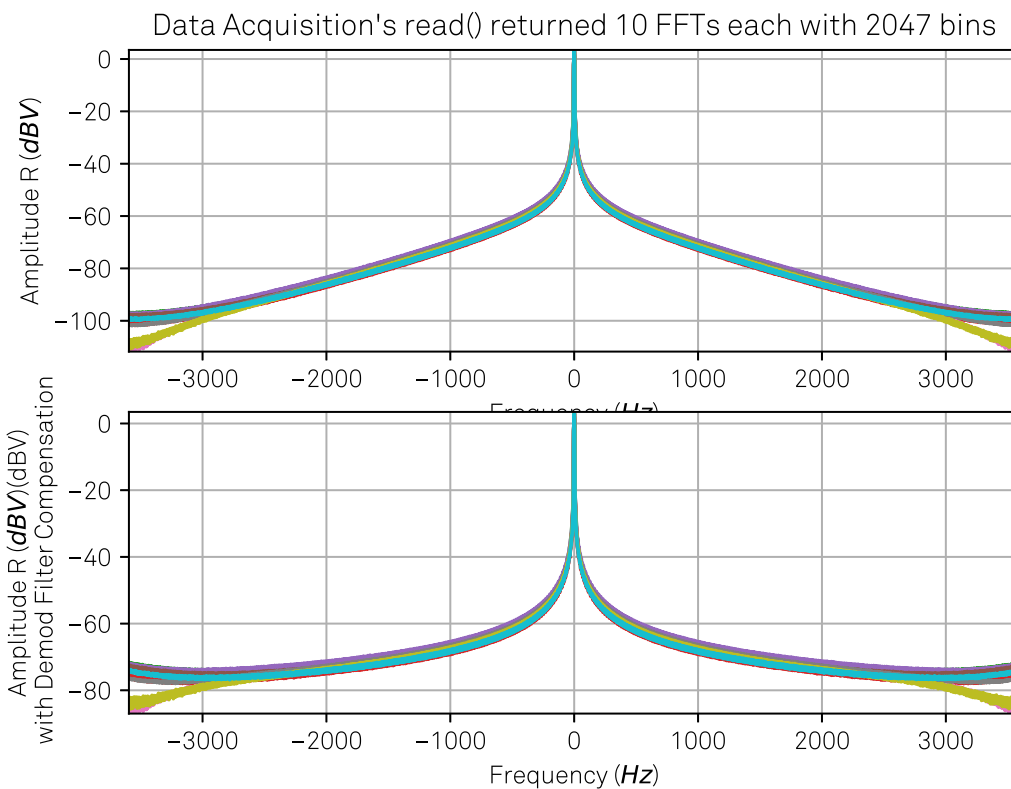


Figure 5.1. The plot produced by the LabOne Python API example demonstrating how to record the Fast Fourier Transform (FFT) applied to demodulator data using the [Data Acquisition Module](#) (`example_data_acquisition_edge_fft.py`).

Exploring which Examples are available

Python's help system can be used to see which examples are available for a particular device class; when help is called on the module the available examples are listed under the "Package Contents" section.

Examples for all Instrument Classes

Here is a list of examples that can run with any instrument classes in the `zhinst.examples.common` package:

```
>>> help('zhinst.examples.common')
```

Help on package `zhinst.examples.common` in `zhinst.examples`:

NAME

`zhinst.examples.common` - Zurich Instruments LabOne Python API Examples (for any instrument class).

PACKAGE CONTENTS

```
example_autoranging_impedance
example_connect
example_connect_config
example_data_acquisition_continuous
example_data_acquisition_edge
example_data_acquisition_edge_fft
example_data_acquisition_grid
example_data_acquisition_trackingedge
example_multidevice_data_acquisition
example_multidevice_sweep
example_pid_advisor_pll
example_poll
example_poll_impedance
example_save_device_settings_expert
example_save_device_settings_simple
example_scope
example_scope_dig_dualchannel
example_scope_dig_segmented
example_scope_dig_stream
example_sweeper
```

DATA

```
__all__ = ['example_autoranging_impedance', 'example_connect', 'exampl...
```

FILE

```
/home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/examples/
common/__init__.py
```

Examples for HDAWG Instruments

Here is a list of the examples available for HDAWG Instruments in the `zhinst.examples.hdawg` package:

```
>>> help('zhinst.examples.hdawg')
```

Help on package `zhinst.examples.hdawg` in `zhinst.examples`:

NAME

`zhinst.examples.hdawg` - Zurich Instruments LabOne Python API Examples for the HDAWG.

PACKAGE CONTENTS

```
example_awg
example_awg_sourcefile
```

DATA

```
__all__ = ['example_awg', 'example_awg_sourcefile']
```

FILE

```
/home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/examples/hdawg/
__init__.py
```

Examples for HF2 Instruments

Here is a list of the examples available for HF2 Instruments in the `zhinst.examples.hf2` package:

```
>>> help('zhinst.examples.hf2')
```

Help on package `zhinst.examples.hf2` in `zhinst.examples`:

NAME

`zhinst.examples.hf2` - Zurich Instruments LabOne Python API Examples for the HF2 Lock-in Amplifier.

PACKAGE CONTENTS

`example_pid_advisor_pll`
`example_scope`

DATA

`__all__` = ['`example_pid_advisor_pll`', '`example_scope`']

FILE

`/home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/examples/hf2/`
`__init__.py`

Examples for UHF Instruments

Here is a list of the examples available for UHF Instruments in the `zhinst.examples.uhf` package:

```
>>> help('zhinst.examples.uhf')
```

Help on package `zhinst.examples.uhf` in `zhinst.examples`:

NAME

`zhinst.examples.uhf` - Zurich Instruments LabOne Python API Examples for the UHF Lock-in Amplifier.

PACKAGE CONTENTS

`example_awg`
`example_awg_sourcefile`
`example_boxcar`

DATA

`__all__` = ['`example_awg`', '`example_awg_sourcefile`', '`example_boxcar`']

FILE

`/home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/examples/uhf/`
`__init__.py`

Locating the **zhinst** Installation Folder and Examples

The examples distributed with the `zhinst` package can serve as a starting point to program your own measurement needs. The example python files, however, are generally not installed in user space. In order to ensure that you have sufficient permission to edit the examples and that your modifications are not overwritten by a later upgrade of the `zhinst` package, please copy them to your own user space before editing them.

The examples are contained in a subfolder of the zhinst package installation folder

```
[PYTHONROOT]\lib\site-packages\zhinst\
```

If you are unsure about the location of your PYTHONROOT, the `__path__` attribute of the `zhinst` module can be used in order to determine its location, for example,

```
from __future__ import print_function
import zhinst
print zhinst.__path__
```

will output something similar to:

```
C:\Python27\lib\site-packages\zhinst
```

5.2.4. Using ziCore Modules in the LabOne Python API

In the LabOne Python API [ziCore Modules](#) are configured and controlled via an instance of the Module's class. This Module object is created using the relevant function from `ziPython.ziDAQServer`. For example, an instance of the [Sweeper Module](#) is created using `ziPython.ziDAQServer`'s `sweep()` function. As such, an API session must be instantiated first using `ziPython.ziDAQServer` (see [Section 1.4.1](#) for more information about initializing API session) and then a sweeper object is created from instance of the API session as following:

```
>>> daq = ziPython.ziDAQServer('localhost', 8004, 6) # Create a connection to the
# Data Server ('localhost' means the Server is running on the same PC as the
# API client, use the device serial of the form 'mf-dev3000' if using an MF
Instrument.
>>> sweeper = daq.sweep();
```

Note, that since creating a Module object without an API connection to the Data Server does not make sense, the Sweeper object is instantiated via the `sweep` method of the `ziDAQServer` class, not directly from the `SweeperModule` class.

The Module's parameters are configured using the Module's `set` method and specifying a path, value pair, for example:

```
>>> sweeper.set('sweep/start', 1.2e5);
```

The parameters can be read-back using the `get` method, which supports wildcards, for example:

```
>>> sweep_params = sweeper.get('sweep/*');
```

The variable `sweep_params` now contains a dictionary of all the Sweeper's parameters. The other main Module commands are similarly used, e.g., `sweeper.execute()`, to start the sweeper. See [Section 3.1.2](#) for more help with Modules and a description of their parameters.

5.2.5. Enabling Logging in the LabOne Python API

Logging from the API is not enabled by default upon initializing a server session with `ziPython`, it must be enabled (after using `connect`) with the `setDebugLevel` command. For example,

```
>>> daq.setDebugLevel(0)
```

sets the API's logging level to 0, which provides the most verbose logging output. The other log levels are defined as following:

```
trace:0, info:1, debug:2, warning:3, error:4, fatal:5, status:6.
```

It is also possible for the user to write their own messages directly to `ziPython` with `writeDebugLog` using the `writeDebugLog` command. For example to write a log message of `info` severity level:

```
>>> daq.writeDebugLog(1, 'Hello log!')
```

On Windows the logs can be found by navigating to the Zurich Instruments "Logs" folder entry in the Windows Start Menu: Programs → Zurich Instruments → LabOne Servers → Logs. This will open an Explorer window displaying folders containing log files from various LabOne components, in particular, the `ziPythonLog` folder contains logs from the LabOne Python API. On Linux, the logs can be found at `/tmp/ziPythonLog_USERNAME`, where `USERNAME` is the same as the output of the `"whoami"` command.

5.3. LabOne Python API Tips and Tricks

In this section some tips and tricks for working with the LabOne Python API are provided.

Data Structures returned by **ziPython**.

The output arguments that **ziPython** returns are designed to use the native data structures that Python users are familiar with and that reflect the data's location in the instruments node hierarchy. For example, when the `poll` command returns data from the instruments fourth demodulator (located in the node hierarchy as `/dev123/demods/3/sample`), the output argument contains a tree of nested dictionaries in which the data can be accessed by

```
data = daq.poll( poll_length, poll_timeout);
x = data['dev123']['demods']['4']['sample']['x'];
y = data['dev123']['demods']['4']['sample']['y'];
```

Tell **poll** to return a flat dictionary

By default, the data returned by `poll` is contained in a tree of nested dictionaries that closely mimics the tree structure of the instrument node hierarchy. By setting the optional fifth argument of `poll` to `True`, the data will be a flat dictionary. This can help avoid many nested `if` statements in order to check that the expected data was returned by `poll`. For example:

```
daq.subscribe('/dev123/demods/0/sample')
flat_dictionary_key = False
data = daq.poll(0.1, 200, 1, flat_dictionary_key)
if 'dev123' in data:
    if 'demods' in data['device']:
        if '0' in data['device']['demods']:
            # access the demodulator data:
            x = data['dev123']['demods']['0']['sample']['x']
            y = data['dev123']['demods']['0']['sample']['y']
```

Could be rewritten more concisely as:

```
daq.subscribe('/dev123/demods/0/sample')
flat_dictionary_key = True
data = daq.poll(0.1, 200, 1, flat_dictionary_key)
if '/dev123/demods/0/sample' in data:
    # access the demodulator data:
    x = data['/dev123/demods/0/sample']['x']
    y = data['/dev123/demods/0/sample']['y']
```

Use the Utility Routines to load Data saved from the LabOne UI and ziControl in Python.

The utilities package `zhinst.utils` contains several routines to help loading `.csv` or `.mat` files saved from either the LabOne User Interface or `ziControl` into Python. These functions are generally minimal wrappers around `NumPy` (`genfromtxt()`) or `scipy` (`loadmat()`) routines. However, the function `load_labone_demod_csv()` is optimized to load demodulator data saved in `.csv` format by the LabOne UI (since it specifies the `.csv` columns' dtypes explicitly) and the function `load_zicontrol_zibin()` can directly load data saved in binary format from `ziControl`. See [Section 5.4.2](#) for reference documentation on these commands.

5.4. LabOne Python API (ziPython) Command Reference

The following reference documentation for ziPython is available in from within a python session using python's help (see [Section 5.2.2](#)) command; It is included here for convenience.

The documentation is grouped by module and class as following:

- [Help for the zhinst Python Package](#)
- [Help for zhinst's Utility Functions](#)
- [Help for ziPython's ziDiscovery class](#)
- [Help for ziPython's ziDAQServer class](#)
- [Help for the AwgModule class](#)
- [Help for the DataAcquisitionModule class](#)
- [Help for the DeviceSettingsModule class](#)
- [Help for the ImpedanceModule class](#)
- [Help for the MultiDeviceSyncModule class](#)
- [Help for the PidAdvisorModule class](#)
- [Help for the ScopeModule class](#)
- [Help for the SweeperModule class](#)

The following two modules will be marked as deprecated in a future release, they are however currently still maintained:

- [Help for the RecorderModule class](#)
- [Help for the ZoomFFTModule class](#)

5.4.1. Help for the zhinst Python Package

```
>>> help('zhinst')

Help on package zhinst:

NAME
    zhinst - Zurich Instruments LabOne Python API

DESCRIPTION
    Contains the API driver, utility functions and examples for Zurich Instruments
    devices.

PACKAGE CONTENTS
    examples (package)
    utils
    ziPython

DATA
    __all__ = ['ziPython', 'utils']

FILE
    /home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/__init__.py
```

5.4.2. Help for zhinst's Utility Functions

```
>>> help('zhinst.utils')
```

Help on module zhinst.utils in zhinst:

NAME

zhinst.utils - Zurich Instruments LabOne Python API Utility Functions.

DESCRIPTION

This module provides basic utility functions for:

- Creating an API session by connecting to an appropriate Data Server.
- Detecting devices.
- Loading and saving device settings.
- Loading data saved by either the Zurich Instruments LabOne User Interface or ziControl into Python as numpy structured arrays.

FUNCTIONS

api_server_version_check(daq)

Issue a warning and return False if the release version of the API used in the session (daq) does not have the same release version as the Data Server (that the API is connected to). If the versions match return True.

Args:

daq (ziDAQServer): An instance of the ziPython.ziDAQServer class (representing an API session connected to a Data Server).

Returns:

Bool: Returns True if the versions of API and Data Server match, otherwise returns False.

autoConnect(default_port=None, api_level=None)

Try to connect to a Zurich Instruments Data Server with an attached available UHF or HF2 device.

Important: autoConnect() does not support MFLI devices.

Args:

default_port (int, optional): The default port to use when connecting to the Data Server (specify 8005 for the HF2 Data Server and 8004 for the UHF Data Server).

api_level (int, optional): The API level to use, either 1, 4 or 5. HF2 only supports Level 1, Level 5 is recommended for UHF and MFLI devices.

Returns:

ziDAQServer: An instance of the ziPython.ziDAQServer class that is used for communication to the Data Server.

Raises:

RuntimeError: If no running Data Server is found or no device is found that is attached to a Data Server.x

If default_port is not specified (=None) then first try to connect to a HF2, if no server devices are found then try to connect to an UHF. This behaviour is useful for the API examples. If we cannot connect to a server and/or

detect a connected device raise a `RunTimeError`.

If `default_port` is 8004 try to connect to a UHF; if it is 8005 try to connect to an HF2. If no server and device is detected on this port raise a `RunTimeError`.

`autoDetect(daq, exclude=None)`

Return a string containing the first device ID (not in the `exclude` list) that is attached to the Data Server connected via `daq`, an instance of the `ziPython.ziDAQServer` class.

Args:

`daq (ziDAQServer)`: An instance of the `ziPython.ziDAQServer` class (representing an API session connected to a Data Server).

`exclude (list of str, optional)`: A list of strings specifying devices to exclude. `autoDetect()` will not return the name of a device in this list.

Returns:

A string specifying the first device ID not in `exclude`.

Raises:

`RunTimeError`: If no device was found.

`RunTimeError`: If `daq` is not an instance of `ziPython.ziDAQServer`.

Example:

```
zhinst.utils
daq = zhinst.utils.autoConnect()
device = zhinst.utils.autoDetect(daq)
```

`bw2tc(bandwidth, order)`

Convert the demodulator 3 dB bandwidth to its equivalent timeconstant for the specified demodulator order.

Inputs:

`bandwidth (double)`: The demodulator 3dB bandwidth to convert.

`order (int)`: The demodulator order (1 to 8) for which to convert the bandwidth.

Output:

`timeconstant (double)`: The equivalent demodulator timeconstant.

`bwtc_scaling_factor(order)`

Return the appropriate scaling factor for bandwidth to timeconstant conversion for the provided demodulator order.

`check_for_sampleloss(timestamps)`

Check whether timestamps are equidistantly spaced, if not, it is an indication that sampleloss has occurred whilst recording the demodulator data.

This function assumes that the timestamps originate from continuously saved demodulator data, during which the demodulator sampling rate was not changed.

Arguments:

`timestamp (numpy array)`: a 1-dimensional array containing demodulator timestamps

Returns:

`idx` (numpy array): a 1-dimensional array indicating the indices in timestamp where sampleloss has occurred. An empty array is returned in no sampleloss was present.

`create_api_session(device_serial, maximum_supported_apilevel, required_devtype='.*', required_options=None, required_err_msg='')`
Create an API session for the specified device.

Args:

`device_serial` (str): A string specifying the device serial number. For example, 'uhf-dev2123' or 'dev2123'.

`maximum_supported_apilevel` (int): The maximum API Level that is supported by the code where the returned API session will be used. The maximum API Level you may use is defined by the device class. HF2 only supports API Level 1 and other devices support API Level 5. You should try to use the maximum level possible to enable extended API features.

`required_devtype` (str): The required device type, e.g., 'HF2LI' or 'MFLI'. This is given by the value of the device node '/devX/features/devtype' or the 'devicetype' discovery property. Raise an exception if the specified device_serial's devtype does not match the 'required_devtype'.

`required_options` (list of str|None): The required device option set. E.g., ['MF', 'PID']. This is given by the value of the device node '/devX/features/options' or the 'options' discovery property. Raise an exception if the specified device_serial's option set does not contain the 'required_options'.

`required_error_msg` (str) : An additional error message to print if either the device specified by the 'device_serial' is not the 'required_devtype' or does not have the 'required_options'.

Returns:

`daq` (ziDAQServer): An instance of the ziPython.ziDAQServer class (representing an API session connected to a Data Server).

`device` (str): The device's ID, this is the string that specifies the device's node branch in the data server's node tree.

`props` (dict): The device's discovery properties as returned by the ziDiscovery get() method.

`default_output_mixer_channel(discovery_props, output_channel=0)`
Return an instrument's default output mixer channel based on the specified 'devicetype' and 'options' discovery properties and the hardware output channel.

This utility function is used by the ziPython examples and returns a node available under the /devX/sigouts/0/{amplitudes,enables}/ branches.

Args:

`discovery_props` (dict): A device's discovery properties as returned by ziDiscovery's get() method.

`output_channel` (int, optional): The zero-based index of the hardware output channel for which to return an output mixer channel.

Returns:

`output_mixer_channel (int)`: The zero-based index of an available signal output mixer channel.

Raises:

Exception: If an invalid signal input index was provided.

`devices(daq)`

Return a list of strings containing the device IDs that are attached to the Data Server connected via `daq`, an instance of the `ziPython.ziDAQServer` class. Returns an empty list if no devices are found.

Args:

`daq (ziDAQServer)`: An instance of the `ziPython.ziDAQServer` class (representing an API session connected to a Data Server).

Returns:

A list of strings of connected device IDs. The list is empty if no devices are detected.

Raises:

`RuntimeError`: If `daq` is not an instance of `ziPython.ziDAQServer`.

Example:

```
import zhinst.utils
daq = zhinst.utils.autoConnect() # autoConnect not supported for MFLI
devices
    device = zhinst.utils.autoDetect(daq)
```

`disable_everything(daq, device)`

Put the device in a known base configuration: disable all extended functionality; disable all streaming nodes.

Output:

`settings (list)`: A list of lists as provided to `ziDAQServer`'s `set()` command. Each sub-list forms a nodepath, value pair. This is a list of nodes configured by the function and may be reused.

Warning: This function is intended as a helper function for the API's examples and it's signature or implementation may change in future releases.

`get_default_settings_path(daq)`

Return the default path used for settings by the `ziDeviceSettings` module.

Arguments:

`daq (instance of ziDAQServer)`: A `ziPython` API session.

Returns:

`settings_path (str)`: The default `ziDeviceSettings` path.

`load_labone_csv(fname)`

Load a CSV file containing generic data as saved by the LabOne User Interface into a numpy structured array.

Arguments:

`filename (str)`: The filename of the CSV file to load.

Returns:

`sample (numpy ndarray)`: A numpy structured array of shape `(num_points,)` whose field names correspond to the column names in the first line of the CSV file. `num_points` is the number of lines in the CSV file - 1.

Example:

```
import zhinst.utils
# Load the CSV file of PID error data (node: /dev2004/pids/0/error)
data = zhinst.utils.load_labone_csv('dev2004_pids_0_error_00000.csv')
import matplotlib.pyplot as plt
# Plot the error
plt.plot(data['timestamp'], data['value'])

load_labone_demod_csv(fname, column_names=('chunk', 'timestamp', 'x', 'y',
'freq', 'phase', 'dio', 'trigger', 'auxin0', 'auxin1'))
Load a CSV file containing demodulator samples as saved by the LabOne User
Interface into a numpy structured array.
```

Arguments:

`fname (file or str)`: The file or filename of the CSV file to load.

`column_names (list or tuple of str, optional)`: A list (or tuple) of column names to load from the CSV file. Default is to load all columns.

Returns:

`sample (numpy ndarray)`: A numpy structured array of shape `(num_points,)` whose field names correspond to the column names in the first line of the CSV file. `num_points` is the number of lines in the CSV file - 1.

Example:

```
import zhinst.utils
sample =
zhinst.utils.load_labone_demod_csv('dev2004_demods_0_sample_00000.csv',
('timestamp', 'x', 'y'))
import matplotlib.pyplot as plt
import numpy as np
plt.plot(sample['timestamp'], np.abs(sample['x'] + 1j*sample['y']))
```

`load_labone_mat(filename)`

A wrapper function for loading a MAT file as saved by the LabOne User Interface with `scipy.io`'s `loadmat()` function. This function is included mainly to document how to work with the data structure return by `scipy.io.loadmat()`.

Arguments:

`filename (str)`: the name of the MAT file to load.

Returns:

`data (dict)`: a nested dictionary containing the instrument data as specified in the LabOne User Interface. The nested structure of ```data``` corresponds to the path of the data's node in the instrument's node hierarchy.

Further comments:

The MAT file saved by the LabOne User Interface (UI) is a Matlab V5.0 data file. The LabOne UI saves the specified data using native Matlab data structures in the same format as are returned by commands in the LabOne Matlab API. More specifically, these data structures are nested Matlab structs, the nested structure of which correspond to the location of the data in the instrument's node hierarchy.

Matlab structs are returned by `scipy.io.loadmat()` as dictionaries, the name of the struct becomes a key in the dictionary. However, as for all objects in MATLAB, structs are in fact arrays of structs, where a single struct is an array of shape (1, 1). This means that each (nested) dictionary that is returned (corresponding to a node in node hierarchy) is loaded by `scipy.io.loadmat` as a 1-by-1 array and must be indexed as such. See the ``Example`` section below.

For more information please refer to the following link:
<http://docs.scipy.org/doc/scipy/reference/tutorial/io.html#matlab-structs>

Example:

```
device = 'dev88'
# See ``Further explanation`` above for a comment on the indexing:
timestamp = data[device][0,0]['demods'][0,0]['sample'][0,0]['timestamp'][0]
x = data[device][0,0]['demods'][0,0]['sample'][0,0]['x'][0]
y = data[device][0,0]['demods'][0,0]['sample'][0,0]['y'][0]
import matplotlib.pyplot as plt
import numpy as np
plt.plot(timestamp, np.abs(x + 1j*y))

# If multiple demodulator's are saved, data from the second demodulator,
# e.g., is accessed as following:
x = data[device][0,0]['demods'][0,1]['sample'][0,0]['x'][0]

load_settings(daq, device, filename)
Load a LabOne settings file to the specified device. This function is
synchronous; it will block until loading the settings has finished.
```

Arguments:

`daq` (instance of `ziDAQServer`): A ziPython API session.

`device` (str): The device ID specifying where to load the settings, e.g., 'dev123'.

`filename` (str): The filename of the xml settings file to load. The filename can include a relative or full path.

Raises:

`RuntimeError`: If loading the settings times out.

Examples:

```
import zhinst.utils as utils
daq = utils.autoConnect()
dev = utils.autoDetect(daq)

# Then, e.g., load settings from a file in the current directory:
utils.load_settings(daq, dev, 'my_settings.xml')
# Then, e.g., load settings from the default LabOne settings path:
filename = 'default_ui.xml'
path = utils.get_default_settings_path(daq)
utils.load_settings(daq, dev, path + os.sep + filename)

load_zicontrol_csv(filename, column_names=('t', 'x', 'y', 'freq', 'dio',
'auxin0', 'auxin1'))
Load a CSV file containing demodulator samples as saved by the ziControl
User Interface into a numpy structured array.
```

Arguments:

`filename` (str): The file or filename of the CSV file to load.

`column_names` (list or tuple of str, optional): A list (or tuple) of column names (demodulator sample field names) to load from the CSV file. Default is to load all columns.

Returns:

`sample` (numpy ndarray): A numpy structured array of shape `(num_points,)` whose field names correspond to the field names of a ziControl demodulator sample. `num_points` is the number of lines in the CSV file - 1.

Example:

```
import zhinst.utils
sample = zhinst.utils.load_labone_csv('Freq1.csv', ('t', 'x', 'y'))
import matplotlib.pyplot as plt
import numpy as np
plt.plot(sample['t'], np.abs(sample['x'] + 1j*sample['y']))

load_zicontrol_zibin(filename, column_names=('t', 'x', 'y', 'freq', 'dio',
'auxin0', 'auxin1'))
Load a ziBin file containing demodulator samples as saved by the ziControl
User Interface into a numpy structured array. This is for data saved by
ziControl in binary format.
```

Arguments:

`filename` (str): The filename of the .ziBin file to load.

`column_names` (list or tuple of str, optional): A list (or tuple) of column names to load from the CSV file. Default is to load all columns.

Returns:

`sample` (numpy ndarray): A numpy structured array of shape `(num_points,)` whose field names correspond to the field names of a ziControl demodulator sample. `num_points` is the number of sample points saved in the file.

Further comments:

Specifying a fewer names in ``column\_names`` will not result in a speed-up as all data is loaded from the binary file by default.

Example:

```
import zhinst.utils
sample = zhinst.utils.load_zicontrol_zibin('Freq1.ziBin')
import matplotlib.pyplot as plt
import numpy as np
plt.plot(sample['t'], np.abs(sample['x'] + 1j*sample['y']))

save_settings(daq, device, filename)
Save settings from the specified device to a LabOne settings file. This
function is synchronous; it will block until saving the settings has
finished.
```

Arguments:

`daq` (instance of ziDAQServer): A ziPython API session.

`device` (str): The device ID specifying where to load the settings, e.g., 'dev123'.

`filename` (str): The filename of the LabOne xml settings file. The filename can include a relative or full path.

Raises:

RuntimeError: If saving the settings times out.

Examples:

```
import zhinst.utils as utils
daq = utils.autoConnect()
dev = utils.autoDetect(daq)

# Then, e.g., save settings to a file in the current directory:
utils.save_settings(daq, dev, 'my_settings.xml')

# Then, e.g., save settings to the default LabOne settings path:
filename = 'my_settings_example.xml'
path = utils.get_default_settings_path(daq)
utils.save_settings(daq, dev, path + os.sep + filename)
```

signin_autorange(daq, device, in_channel)
Perform an automatic adjustment of the signal input range based on the measured input signal. This utility function starts the functionality implemented in the device's firmware and waits until it has completed. The range is set by the firmware based on the measured input signal's amplitude measured over approximately 100 ms.

Requirements:

A devtype that supports autorange functionality on the firmware level, e.g., UHFLI, MFLI, MFIA.

Arguments:

daq (instance of ziDAQServer): A ziPython API session.

device (str): The device ID on which to perform the signal input autorange.

in_channel (int): The index of the signal input channel to autorange.

Raises:

AssertionError: If the functionality is not supported by the device or an invalid `in_channel` was specified.

RuntimeError: If autorange functionality does not complete within the timeout.

Example:

```
import zhinst.utils
device_serial = 'dev2006'
(daq, _, _) = zhinst.utils.create_api_session(device_serial, 5)
input_channel = 0
zhinst.utils.signin_autorange(daq, device_serial, input_channel)
```

systemtime_to_datetime(systemtime)
Convert the LabOne "systemtime" returned in LabOne data headers from microseconds since Unix epoch to a datetime object with microsecond precision.

Example:

```
import zhinst.examples as ziex
import zhinst.utils as ziutils
data = ziex.common.example_sweeper.run_example('dev2006')
systemtime = data[0][0]['header']['systemtime'][0]
t_datetime = ziutils.systemtime_to_datetime(systemtime)
t_datetime.strftime('%Y-%m-%dT%H:%M:%S.%f')
```

tc2bw(timeconstant, order)

Convert the demodulator timeconstant to its equivalent 3 dB bandwidth for the specified demodulator order.

Inputs:

timeconstant (double): The equivalent demodulator timeconstant.

order (int): The demodulator order (1 to 8) for which to convert the bandwidth.

Output:

bandwidth (double): The demodulator 3dB bandwidth to convert.

DATA

```
LABONE_DEMOD_DTYPE = [('chunk', 'u8'), ('timestamp', 'u8'), ('x', 'f8'...
LABONE_DEMOD_FORMATS = ('u8', 'u8', 'f8', 'f8', 'f8', 'f8', 'u4', 'u4'...
LABONE_DEMOD_NAMES = ('chunk', 'timestamp', 'x', 'y', 'freq', 'phase',...
ZICONTROL_DTYPE = [('t', 'f8'), ('x', 'f8'), ('y', 'f8'), ('freq', 'f8'...
ZICONTROL_FORMATS = ('f8', 'f8', 'f8', 'f8', 'u4', 'f8', 'f8')
ZICONTROL_NAMES = ('t', 'x', 'y', 'freq', 'dio', 'auxin0', 'auxin1')
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0)...
```

FILE

```
/home/ci/.pyenv/versions/3.7.1/lib/python3.7/site-packages/zhinst/utils.py
```

5.4.3. Help for ziPython's ziDiscovery class

```
>>> help('zhinst.ziPython.ziDiscovery')
```

Help on class ziDiscovery in zhinst.ziPython:

```
zhinst.ziPython.ziDiscovery = class ziDiscovery(Boost.Python.instance)
|   Class to find devices and get their connectivity properties.
|
|   Method resolution order:
|       ziDiscovery
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       __init__( (object)arg1) -> None
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   find(...)
|       find( (ziDiscovery)arg1, (str)arg2) -> str :
|           Return the device id for a given device address.
|           arg1: Device address string e.g. UHF-DEV2000.
|
|   findAll(...)
|       findAll( (ziDiscovery)arg1) -> list :
|           Return a list of all discoverable devices.
|
|   get(...)
|       get( (ziDiscovery)arg1, (str)arg2) -> object :
|           Return the device properties for a given device id.
|           arg1: Device id string e.g. DEV2000.
|
|   setDebugLevel(...)
|       setDebugLevel( (ziDiscovery)arg1, (int)arg2) -> None :
|           Set debug level.
```



```
        arg1: debug level.

-----
Data and other attributes defined here:

__instance_size__ = 64

-----
Static methods inherited from Boost.Python.instance:

__new__(*args, **kwargs) from Boost.Python.class
    Create and return a new object.  See help(type) for accurate signature.

-----
Data descriptors inherited from Boost.Python.instance:

__dict__
__weakref__
```

5.4.4. Help for ziPython's **ziDAQServer** class

```
>>> help('zhinst.ziPython.ziDAQServer')

Help on class ziDAQServer in zhinst.ziPython:

zhinst.ziPython.ziDAQServer = class ziDAQServer(Boost.Python.instance)
|   Class to connect with a Zurich Instruments data server.
|
|   Method resolution order:
|       ziDAQServer
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       __init__( (object)arg1) -> None
|
|       __init__( (object)arg1, (str)arg2, (int)arg3) -> None :
|           Connect to the server by using host address and port number.
|           arg1: Reference to the ziDAQServer class.
|           arg2: Host string e.g. '127.0.0.1' for localhost.
|           arg3: Port number e.g. 8004 for the ziDataServer.
|
|       __init__( (object)arg1, (str)arg2, (int)arg3, (int)arg4) -> None :
|           Connect to the server by using host address and port number.
|           arg1: Reference to the ziDAQServer class.
|           arg2: Host string e.g. '127.0.0.1' for localhost.
|           arg3: Port number e.g. 8004 for the ziDataServer.
|           arg4: API level number.
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   asyncSetDouble(...)
|       asyncSetDouble( (ziDAQServer)arg1, (str)arg2, (float)arg3) -> None :
|           Use with care: returns immediately, any errors silently ignored.
|           arg1: Reference to the ziDAQServer class.
|           arg2: Path string of the node.
|
|   asyncSetInt(...)
|       asyncSetInt( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> None :
|           Use with care: returns immediately, any errors silently ignored.
|           arg1: Reference to the ziDAQServer class.
|           arg2: Path string of the node.
```

```
asyncSetString(...)
    asyncSetString( (ziDAQServer)arg1, (str)arg2, (object)arg3) -> None :
        Use with care: returns immediately, any errors silently ignored.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

awgModule(...)
    awgModule( (ziDAQServer)arg1) -> AwgModule :
        Create a AwgModule class. This will start a thread for running an
        asynchronous AwgModule.
        arg1: Reference to the ziDAQServer class.

connect(...)
    connect( (ziDAQServer)arg1) -> None

connectDevice(...)
    connectDevice( (ziDAQServer)arg1, (str)arg2, (str)arg3, (str)arg4) -> None :
        Connect with the data server to a specified device over the specified
        interface. The device must be visible to the server. If the device is
        already connected the call will be ignored. The function will block
        until the device is connected and the device is ready to use. This
        method is useful for UHF devices offering several communication
        interfaces.
        arg1: Reference to the ziDAQServer class.
        arg2: Device serial.
        arg3: Device interface.
        arg4: Optional interface parameters string.

    connectDevice( (ziDAQServer)arg1, (str)arg2, (str)arg3) -> None

dataAcquisitionModule(...)
    dataAcquisitionModule( (ziDAQServer)arg1) -> DataAcquisitionModule :
        Create a DataAcquisitionModule class. This will start a thread for
        running an asynchronous Data Acquisition Module.
        arg1: Reference to the ziDAQServer class.

deviceSettings(...)
    deviceSettings( (ziDAQServer)arg1) -> DeviceSettingsModule :
        Create a DeviceSettingsModule class. This will start a thread for running
        an asynchronous DeviceSettingsModule.
        arg1: Reference to the ziDAQServer class.
        arg2: Timeout in [ms]. Recommended value is 500ms.
        DEPRECATED, ignored

    deviceSettings( (ziDAQServer)arg1, (int)arg2) -> DeviceSettingsModule

disconnect(...)
    disconnect( (ziDAQServer)arg1) -> None

disconnectDevice(...)
    disconnectDevice( (ziDAQServer)arg1, (str)arg2) -> None :
        Disconnect a device on the data server. This function will return
        immediately. The disconnection of the device may not yet finished.
        arg1: Reference to the ziDAQServer class.
        arg2: Device serial string of device to disconnect.

echoDevice(...)
    echoDevice( (ziDAQServer)arg1, (str)arg2) -> None :
        Sends an echo command to a device and blocks until
        answer is received. This is useful to flush all
        buffers between API and device to enforce that
        further code is only executed after the device executed
        a previous command.
        arg1: Reference to the ziDAQServer class.
        arg2: Device string e.g. 'dev100'.
```

```

| flush(...)
|     flush( (ziDAQServer)arg1) -> None :
|         Flush all data in the socket connection and API buffers.
|         Call this function before a subscribe with subsequent poll
|         to get rid of old streaming data that might still be in
|         the buffers.
|         arg1: Reference to the ziDAQServer class.
|
| get(...)
|     get( (ziDAQServer)arg1, (str)arg2, (bool)arg3, (int)arg4) -> object :
|         Return a dict with all nodes from the specified sub-tree.
|         High-speed streaming nodes (e.g. /devN/demods/0/sample)
|         are not returned. Wildcards (*) may be used, in which case
|         read-only nodes are ignored.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string of the node. Use wild card to
|             select all.
|         arg3[optional]: Specify which type of data structure to return.
|             Return data either as a flat dict (True) or as a nested
|             dict tree (False). Default = False.
|         arg4[optional]: Specify which type of nodes to include in the
|             result. Allowed:
|             ZI_LIST_NODES_SETTINGSONLY = 8 (default)
|             ZI_LIST_NODES_NONE = 0 (all nodes)
|
|     get( (ziDAQServer)arg1, (str)arg2 [, (bool)arg3]) -> object
|
| getAsEvent(...)
|     getAsEvent( (ziDAQServer)arg1, (str)arg2) -> None :
|         Trigger an event on the specified node. The node data is returned by a
|         subsequent poll command.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string of the node.
|
| getAuxInSample(...)
|     getAuxInSample( (ziDAQServer)arg1, (str)arg2) -> object :
|         Returns a single auxin sample. The auxin data is averaged in contrast to
|         the auxin data embedded in the demodulator sample.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string
|
| getByte(...)
|     getByte( (ziDAQServer)arg1, (str)arg2) -> object :
|         Get a byte array (string) value from the specified node.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string of the node.
|
| getComplex(...)
|     getComplex( (ziDAQServer)arg1, (str)arg2) -> complex :
|         Get a complex double value from the specified node.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string of the node.
|
| getConnectionAPILevel(...)
|     getConnectionAPILevel( (ziDAQServer)arg1) -> int :
|         Returns ziAPI level used for the active connection.
|
| getDIO(...)
|     getDIO( (ziDAQServer)arg1, (str)arg2) -> object :
|         Returns a single DIO sample.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Path string
|
| getDebugLogpath(...)
|     getDebugLogpath( (ziDAQServer)arg1) -> str :
|         Returns the path where logfiles are stored. Note, it will return
|         the empty string if logging has not been enabled via setDebugLevel().

```

```

        arg1: Reference to the ziDAQServer class.

getDouble(...)
    getDouble( (ziDAQServer)arg1, (str)arg2) -> float :
        Get a double value from the specified node.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

getInt(...)
    getInt( (ziDAQServer)arg1, (str)arg2) -> int :
        Get a integer value from the specified node.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

getLastError(...)
    getLastError( (ziDAQServer)arg1) -> object :
        Return the last error message of the API.
        arg1: Reference to the ziDAQServer class.

getList(...)
    getList( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> object :
        Return a list with all nodes from the specified sub-tree.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node. Use wild card to
            select all.
        arg3[optional]: Specify which type of nodes to include in the
            result. Allowed:
            ZI_LIST_NODES_SETTINGSONLY = 0x08 (default)
            ZI_LIST_NODES_NONE = 0x00 (all nodes)

    getList( (ziDAQServer)arg1, (str)arg2) -> object

getSample(...)
    getSample( (ziDAQServer)arg1, (str)arg2) -> object :
        Returns a single demodulator sample (including DIO and AuxIn). For more
        efficient data recording use subscribe and poll methods.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string

getString(...)
    getString( (ziDAQServer)arg1, (str)arg2) -> object :
        Get a string value from the specified node.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

getStringUnicode(...)
    getStringUnicode( (ziDAQServer)arg1, (str)arg2) -> object :
        Get a unicode string value from the specified node.
        The returned string is unicode encoded.
        Only relevant for Python versions older than V3.0.
        For Python versions 3.0 and later, getString can be used instead.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

help(...)
    help( (ziDAQServer)arg1, (str)arg2) -> None :
        Returns a well-formatted description of a node. Only UHF and MF devices
        support this functionality.
        arg1: Reference to the ziDAQServer class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.

impedanceModule(...)
    impedanceModule( (ziDAQServer)arg1) -> ImpedanceModule :
        Create a ImpedanceModule class. This will start a thread for
        running an asynchronous ImpedanceModule.

```

```

        arg1: Reference to the ziDAQServer class.

listNodes(...)
    listNodes( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the ziDAQServer class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            ziPython.ziListEnum.none -> 0x00
                The default flag, returning a simple
                listing of the given node
            ziPython.ziListEnum.recursive -> 0x01
                Returns the nodes recursively
            ziPython.ziListEnum.absolute -> 0x02
                Returns absolute paths
            ziPython.ziListEnum.leafsonly -> 0x04
                Returns only nodes that are leafs,
                which means the they are at the
                outermost level of the tree.
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
            ziPython.ziListEnum.streamingonly -> 0x10
                Returns only streaming nodes
            ziPython.ziListEnum.subscribedonly -> 0x20
                Returns only subscribed nodes
            ziPython.ziListEnum.basechannel -> 0x40
                Return only one instance of a node in case of multiple
                channels
            Or any combination of flags can be used.

listNodesJSON(...)
    listNodesJSON( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> str :
        Returns a list of nodes with description found at the specified path.
        Only UHF and MF devices support this functionality.
        arg1: Reference to the ziDAQServer class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            They are the same as for listNodes(), except that
            0x01, 0x02 and 0x04 are enforced
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
            ziPython.ziListEnum.streamingonly -> 0x10
                Returns only streaming nodes
            ziPython.ziListEnum.subscribedonly -> 0x20
                Returns only subscribed nodes
            ziPython.ziListEnum.basechannel -> 0x40
                Return only one instance of a node in case of multiple
                channels
            Or any combination of flags can be used.

logOff(...)
    logOff( (ziDAQServer)arg1) -> None :
        Disables logging of commands sent to a server.
        arg1: Reference to the ziDAQServer class.

logOn(...)
    logOn( (ziDAQServer)arg1, (int)arg2, (str)arg3, (int)arg4) -> None :
        Enables logging of commands sent to a server.
        arg1: Reference to the ziDAQServer class.
        arg2: Flags (LOG_NONE:          0x00000000
            LOG_SET_DOUBLE:          0x00000001

```

```

|                                     LOG_SET_INT:          0x00000002
|                                     LOG_SET_BYTE:         0x00000004
|                                     LOG_SET_STRING:       0x00000008
|                                     LOG_SYNC_SET_DOUBLE:  0x00000010
|                                     LOG_SYNC_SET_INT:     0x00000020
|                                     LOG_SYNC_SET_BYTE:    0x00000040
|                                     LOG_SYNC_SET_STRING:  0x00000080
|                                     LOG_GET_DOUBLE:      0x00000100
|                                     LOG_GET_INT:         0x00000200
|                                     LOG_GET_BYTE:       0x00000400
|                                     LOG_GET_STRING:     0x00000800
|                                     LOG_GET_DEMOD:      0x00001000
|                                     LOG_GET_DIO:        0x00002000
|                                     LOG_GET_AUXIN:      0x00004000
|                                     LOG_GET_COMPLEX:    0x00008000
|                                     LOG_LISTNODES:      0x00010000
|                                     LOG_SUBSCRIBE:      0x00020000
|                                     LOG_UNSUBSCRIBE:    0x00040000
|                                     LOG_GET_AS_EVENT:   0x00080000
|                                     LOG_UPDATE:         0x00100000
|                                     LOG_POLL_EVENT:     0x00200000
|                                     LOG_POLL:          0x00400000
|                                     LOG_ALL :          0xffffffff
|
|     arg3: Log file name.
|     arg4: Log style (LOG_STYLE_TELNET: 0 (default),
|             LOG_STYLE_MATLAB: 1, LOG_STYLE_PYTHON: 2).
|
|     logOn( (ziDAQServer)arg1, (int)arg2, (str)arg3) -> None
|
| multiDeviceSyncModule(...)
|     multiDeviceSyncModule( (ziDAQServer)arg1) -> MultiDeviceSyncModule :
|         Create a MultiDeviceSyncModule class. This will start a thread for
|         running an asynchronous MultiDeviceSync module.
|         arg1: Reference to the ziDAQServer class.
|
| pidAdvisor(...)
|     pidAdvisor( (ziDAQServer)arg1) -> PidAdvisorModule :
|         Create a PidAdvisorModule class. This will start a thread for running an
|         asynchronous PidAdvisorModule.
|         arg1: Reference to the ziDAQServer class.
|         arg2: Timeout in [ms]. Recommended value is 500ms.
|             DEPRECATED, ignored
|
|     pidAdvisor( (ziDAQServer)arg1, (int)arg2) -> PidAdvisorModule
|
| poll(...)
|     poll( (ziDAQServer)arg1, (float)arg2, (int)arg3, (int)arg4, (bool)arg5) ->
object :
|         Continuously check for value changes (by calling pollEvent) in all
|         subscribed nodes for the specified duration and return the data. If
|         no value change occurs in subscribed nodes before duration + timeout,
|         poll returns no data. This function call is blocking (it is
|         synchronous). However, since all value changes are returned since
|         either subscribing to the node or the last poll (assuming no buffer
|         overflow has occurred on the Data Server), this function may be used
|         in a quasi-asynchronous manner to return data spanning a much longer
|         time than the specified duration. The timeout parameter is only
|         relevant when communicating in a slow network. In this case it may be
|         set to a value larger than the expected round-trip time in the
|         network.
|         Poll returns a dict tree containing the recorded data (see arg5).
|         arg1: Reference to the ziDAQServer class.
|         arg2: Recording time in [s]. The function will block during that.
|             time.
|         arg3: Poll timeout in [ms]. Recommended value is 500ms.
|         arg4[optional]: Poll flags.
|             FILL = 0x0001: Fill holes (only possible in

```

```

|                                     combination with FILL).
|                                     THROW = 0x0004: Throw EOFError exception if sample
|                                               loss is detected.
|                                     DETECT = 0x0008: Detect data loss holes.
|                                     arg5[optional]: Specify which type of data structure to return.
|                                               Return data either as a flat dict (True) or as a nested
|                                               dict tree (False). Default = False.
|
| poll( (ziDAQServer)arg1, (float)arg2, (int)arg3 [, (int)arg4]) -> object
|
| pollEvent(...)
|   pollEvent( (ziDAQServer)arg1, (int)arg2) -> object :
|     Return the value changes that occurred in one single subscribed node.
|     This is a low-level function. The poll function is better suited in
|     nearly all cases. To get all data waiting in the buffers this command
|     should be executed continuously until nothing is returned anymore.
|     arg1: Reference to the ziDAQServer class.
|     arg2: Poll timeout in [ms]. Recommended value is 500ms.
|
| programRT(...)
|   programRT( (ziDAQServer)arg1, (str)arg2, (str)arg3) -> None :
|     Program RT.
|     arg1: Device identifier e.g. 'dev99'.
|     arg2: File name of the RT program.
|
| record(...)
|   record( (ziDAQServer)arg1) -> RecorderModule :
|     Create a recording class. This will start a thread for asynchronous
|     recording.
|     arg1: Reference to the ziDAQServer class.
|     arg2: Maximum recording time for single triggers in [s].
|           DEPRECATED, set 'buffersize' param instead.
|     arg3: Timeout in [ms]. Recommended value is 500ms.
|           DEPRECATED, ignored.
|     arg4[optional]: Record flags.
|                     DEPRECATED, set 'flags' param instead.
|                     FILL = 0x0001 : Fill holes.
|                     ALIGN = 0x0002 : Align data that contains a
|                                     timestamp.
|                     THROW = 0x0004 : Throw EOFError exception if
|                                     sample loss is detected.
|                     DETECT = 0x0008: Detect data loss holes.
|
|   record( (ziDAQServer)arg1, (float)arg2, (int)arg3 [, (int)arg4]) ->
RecorderModule
|
| revision(...)
|   revision( (ziDAQServer)arg1) -> int :
|     Get the revision number of the Python interface of Zurich Instruments.
|     arg1: Reference to the ziDAQServer class.
|
| scopeModule(...)
|   scopeModule( (ziDAQServer)arg1) -> ScopeModule :
|     Create a ScopeModule class. This will start a thread for running an
|     asynchronous ScopeModule.
|     arg1: Reference to the ziDAQServer class.
|
| set(...)
|   set( (ziDAQServer)arg1, (object)arg2) -> None :
|     Set multiple nodes.
|     arg1: Reference to the ziDAQServer class.
|     arg2: A list of path/value pairs.
|
| setByte(...)
|   setByte( (ziDAQServer)arg1, (str)arg2, (object)arg3) -> None :
|     arg1: Reference to the ziDAQServer class.

```

```

        arg2: Path string of the node.

setComplex(...)
    setComplex( (ziDAQServer)arg1, (str)arg2, (complex)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

setDebugLevel(...)
    setDebugLevel( (ziDAQServer)arg1, (int)arg2) -> None :
        Enables debug log and sets the debug level.
        arg1: Reference to the ziDAQServer class.
        arg2: Debug level (trace:0, info:1, debug:2, warning:3, error:4,
            fatal:5, status:6).

setDouble(...)
    setDouble( (ziDAQServer)arg1, (str)arg2, (float)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

setInt(...)
    setInt( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

setLastError(...)
    setLastError( (ziDAQServer)arg1, (str)arg2) -> None :
        Update the last error message with the given error string.
        arg1: Reference to the ziDAQServer class.
        arg2: Error string.

setString(...)
    setString( (ziDAQServer)arg1, (str)arg2, (object)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

subscribe(...)
    subscribe( (ziDAQServer)arg1, (object)arg2) -> None :
        Subscribe to one or several nodes. Fetch data with the poll
        command. In order to avoid fetching old data that is still in the
        buffer execute a flush command before subscribing to data streams.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node. Use wild card to
            select all. Alternatively also a list of path
            strings can be specified.

sweep(...)
    sweep( (ziDAQServer)arg1) -> SweeperModule :
        Create a sweep class. This will start a thread for asynchronous
        sweeping.
        arg1: Reference to the ziDAQServer class.
        arg2: Timeout in [ms]. Recommended value is 500ms.
            DEPRECATED, ignored

    sweep( (ziDAQServer)arg1, (int)arg2) -> SweeperModule

sync(...)
    sync( (ziDAQServer)arg1) -> None :
        Synchronize all data path. Ensures that get and poll
        commands return data which was recorded after the
        setting changes in front of the sync command. This
        sync command replaces the functionality of all syncSet,
        flush, and echoDevice commands.
        arg1: Reference to the ziDAQServer class.

syncSetDouble(...)
    syncSetDouble( (ziDAQServer)arg1, (str)arg2, (float)arg3) -> float :
        arg1: Reference to the ziDAQServer class.

```



```

        arg2: Path string of the node.

syncSetInt(...)
    syncSetInt( (ziDAQServer)arg1, (str)arg2, (int)arg3) -> int :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

syncSetString(...)
    syncSetString( (ziDAQServer)arg1, (str)arg2, (object)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.

unsubscribe(...)
    unsubscribe( (ziDAQServer)arg1, (object)arg2) -> None :
        Unsubscribe data streams. Use this command after recording to avoid
        buffer overflows that may increase the latency of other command.
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node. Use wild card to
            select all. Alternatively also a list of path
            strings can be specified.

update(...)
    update( (ziDAQServer)arg1) -> None :
        Check if additional devices are attached. This function is not needed
        for servers running under windows as devices will be detected
        automatically.
        arg1: Reference to the ziDAQServer class.

vectorWrite(...)
    vectorWrite( (ziDAQServer)arg1, (str)arg2, (object)arg3) -> None :
        arg1: Reference to the ziDAQServer class.
        arg2: Path string of the node.
        arg3: Vector ((u)int8, (u)int16, (u)int32, (u)int64, float, double)
            or string to write.

version(...)
    version( (ziDAQServer)arg1) -> str :
        Get version string of the Python interface of Zurich Instruments.
        arg1: Reference to the ziDAQServer class.

writeDebugLog(...)
    writeDebugLog( (ziDAQServer)arg1, (int)arg2, (str)arg3) -> None :
        Outputs message to the debug log (if enabled).
        arg1: Reference to the ziDAQServer class.
        arg2: Severity (trace:0, info:1, debug:2, warning:3, error:4,
            fatal:5, status:6).
        arg3: Message to output to the log.

zoomFFT(...)
    zoomFFT( (ziDAQServer)arg1) -> ZoomFFTModule :
        Create a ZoomFFTModule class. This will start a thread for running an
        asynchronous ZoomFFTModule.
        arg1: Reference to the ziDAQServer class.
        arg2: Timeout in [ms]. Recommended value is 500ms.
            DEPRECATED, ignored

    zoomFFT( (ziDAQServer)arg1, (int)arg2) -> ZoomFFTModule

-----
Data and other attributes defined here:

__instance_size__ = 88

-----
Static methods inherited from Boost.Python.instance:

__new__(*args, **kwargs) from Boost.Python.class

```

```
|         Create and return a new object.  See help(type) for accurate signature.
|
| -----
| Data descriptors inherited from Boost.Python.instance:
|
| __dict__
|
| __weakref__
```

5.4.5. Help for the **AwgModule** class

An instance of AwgModule is initialized using the awgModule method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.awgModule')
```

Help on built-in function awgModule in zhinst.ziPython.ziDAQServer:

```
zhinst.ziPython.ziDAQServer.awgModule = awgModule(...)
awgModule( (ziDAQServer)arg1) -> AwgModule :
    Create a AwgModule class. This will start a thread for running an
    asynchronous AwgModule.
    arg1: Reference to the ziDAQServer class.
```

Reference help for the AwgModule class.

```
>>> help('zhinst.ziPython.AwgModule')
```

Help on class AwgModule in zhinst.ziPython:

```
zhinst.ziPython.AwgModule = class AwgModule(Boost.Python.instance)
|   Method resolution order:
|       AwgModule
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       Raises an exception
|       This class cannot be instantiated from Python
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   clear(...)
|       clear( (AwgModule)arg1) -> None :
|           End the awgModule thread.
|
|   execute(...)
|       execute( (AwgModule)arg1) -> None :
|           Starts the awgModule if not yet running.
|
|   finish(...)
|       finish( (AwgModule)arg1) -> None :
|           Stop the awgModule.
|
|   finished(...)
|       finished( (AwgModule)arg1) -> bool :
|           Check if the command execution has finished. Returns True if finished.
|
|   get(...)
|       get( (AwgModule)arg1, (str)arg2, (bool)arg3) -> object :
|           Return a dict with all nodes from the specified sub-tree.
|           arg1: Reference to the AwgModule class.
|           arg2: Path string of the node. Use wild card to
|               select all.
```

```

        |         arg3[optional]: Specify which type of data structure to return.
        |         Return data either as a flat dict (True) or as a nested
        |         dict tree (False). Default = False.
        |
        |         get( (AwgModule)arg1, (str)arg2) -> object
        |
        |     getDouble(...)
        |         getDouble( (AwgModule)arg1, (str)arg2) -> float :
        |             Return the floating point double value for the specified path.
        |             arg1: Reference to the AwgModule class.
        |             arg2: Path string of the node.
        |
        |     getInt(...)
        |         getInt( (AwgModule)arg1, (str)arg2) -> int :
        |             Return the integer value for the specified path.
        |             arg1: Reference to the AwgModule class.
        |             arg2: Path string of the node.
        |
        |     getString(...)
        |         getString( (AwgModule)arg1, (str)arg2) -> object :
        |             Return the string value for the specified path.
        |             arg1: Reference to the AwgModule class.
        |             arg2: Path string of the node.
        |
        |     getStringUnicode(...)
        |         getStringUnicode( (AwgModule)arg1, (str)arg2) -> object :
        |             Get a unicode string value from the specified node.
        |             The returned string is unicode encoded.
        |             Only relevant for Python versions older than V3.0.
        |             For Python versions 3.0 and later, getString can be used instead.
        |             arg1: Reference to the AwgModule class.
        |             arg2: Path string of the node.
        |
        |     listNodes(...)
        |         listNodes( (AwgModule)arg1, (str)arg2, (int)arg3) -> list :
        |             This function returns a list of node names found at the specified path.
        |             arg1: Reference to the AwgModule class.
        |             arg2: Path for which the nodes should be listed. The path may
        |                 contain wildcards so that the returned nodes do not
        |                 necessarily have to have the same parents.
        |             arg3: Enum that specifies how the selected nodes are listed.
        |                 ziPython.ziListEnum.none -> 0x00
        |                     The default flag, returning a simple
        |                     listing if the given node
        |                 ziPython.ziListEnum.recursive -> 0x01
        |                     Returns the nodes recursively
        |                 ziPython.ziListEnum.absolute -> 0x02
        |                     Returns absolute paths
        |                 ziPython.ziListEnum.leafsonly -> 0x04
        |                     Returns only nodes that are leafs,
        |                     which means the they are at the
        |                     outermost level of the tree.
        |                 ziPython.ziListEnum.settingsonly -> 0x08
        |                     Returns only nodes which are marked
        |                     as setting
        |             Or combinations of flags can be used.
        |
        |     progress(...)
        |         progress( (AwgModule)arg1) -> object :
        |             Reports the progress of the command with a number between 0 and 1.
        |
        |     save(...)
        |         save( (AwgModule)arg1, (str)arg2) -> None :
        |             Not relevant for the awgModule module.
        |
        |     set(...)
        |         set( (AwgModule)arg1, (str)arg2, (float)arg3) -> None :

```

AWG Parameters		Type	Description
Path name	awg/enable	bool	Start the AWG sequencers. In MDS mode, this will enable all devices in the correct order.
sourcefile	awgModule/compiler/sourcefile	string	AWG sequencer program file to load. The file needs to be saved in the 'src' sub-directory of the AWG settings directory.
	awgModule/compiler/sourcestr	string	AWG sequencer program string to load. Allows to compile a sequencer program without saving it to a file first. Compilation will start directly after setting of the parameter. Setting awgModule/compiler/start is not necessary.
	awgModule/compiler/waveforms	string	Comma-separated list waveform files to be used by the AWG sequencer program.
	awgModule/compiler/status	string	Status message of the compiler (read-only)
	awgModule/compiler/status	int	Status of the compiler (read-only): -1 = idle 0 = compilation successful 1 = compilation failed 2 = compilation encountered warnings
	awgModule/compiler/start	bool	Start compilation from source file specified in awgModule/compiler/sourcefile and upload of the AWG sequencer program to the device. Will be reset after completion or on error.
	awgModule/device	string	Device that should be used to run AWG programs e.g. 'dev99'
	awgModule/directory	string	Directory where AWG sequencer programs, waveforms and ELF files should be located. If not set, the default settings location of the LabOne software is used.
	awgModule/elf/file	string	File name of the ELF file to upload. If not set, the name will be set automatically based on the source file name. The file will be saved in the 'elf' sub-directory of the AWG settings directory.
	awgModule/elf/upload	bool	Start upload of the AWG sequencer program to the device. Will be reset after completion or on error.
	awgModule/elf/status	int	Status of the ELF file upload (read-only) -1 = idle 0 = upload successful 1 = upload failed 2 = upload is in progress
	awgModule/elf/checksum	int	Checksum of the uploaded ELF file (read-only).
	awgModule/progress	double	Reports the progress of the upload with a number between 0 and 1.
set((AwgModule)arg1, (str)arg2, (int)arg3) -> None			
set((AwgModule)arg1, (str)arg2, (str)arg3) -> None			
set((AwgModule)arg1, (object)arg2) -> None :			

```
|         arg1: Reference to the AwgModule class.
|         arg2: A list of path/value pairs.
|
| subscribe(...)
|     subscribe( (AwgModule)arg1, (str)arg2) -> None :
|         Not relevant for the awgModule module.
|
| trigger(...)
|     trigger( (AwgModule)arg1) -> None :
|         Not applicable to this module.
|
| unsubscribe(...)
|     unsubscribe( (AwgModule)arg1, (str)arg2) -> None :
|         Not relevant for the awgModule module.
|
| -----
| Static methods inherited from Boost.Python.instance:
|
| __new__(*args, **kwargs) from Boost.Python.class
|     Create and return a new object. See help(type) for accurate signature.
|
| -----
| Data descriptors inherited from Boost.Python.instance:
|
| __dict__
|
| __weakref__
```

5.4.6. Help for the **DataAcquisitionModule** class

An instance of DataAcquisitionModule is initialized using the dataAcquisitionModule method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.dataAcquisitionModule')

Help on built-in function dataAcquisitionModule in zhinst.ziPython.ziDAQServer:

zhinst.ziPython.ziDAQServer.dataAcquisitionModule = dataAcquisitionModule(...)
    dataAcquisitionModule( (ziDAQServer)arg1) -> DataAcquisitionModule :
        Create a DataAcquisitionModule class. This will start a thread for
        running an asynchronous Data Acquisition Module.
        arg1: Reference to the ziDAQServer class.
```

Reference help for the DataAcquisitionModule class.

```
>>> help('zhinst.ziPython.DataAcquisitionModule')

Help on class DataAcquisitionModule in zhinst.ziPython:

zhinst.ziPython.DataAcquisitionModule = class
    DataAcquisitionModule(Boost.Python.instance)
    | Method resolution order:
    |     DataAcquisitionModule
    |     Boost.Python.instance
    |     builtins.object
    |
    | Static methods defined here:
    |
    | __init__(...)
    |     Raises an exception
    |     This class cannot be instantiated from Python
    |
    | __reduce__ = <unnamed Boost.Python function>(...)
```

```

| clear(...)
|     clear( (DataAcquisitionModule)arg1) -> None :
|         End the acquisition thread.
|
| execute(...)
|     execute( (DataAcquisitionModule)arg1) -> None :
|         Start the data acquisition. After that command any
|         trigger will start the measurement.
|         Subscription or unsubscription is not
|         possible until the acquisition is finished.
|
| finish(...)
|     finish( (DataAcquisitionModule)arg1) -> None :
|         Stop acquisition. The acquisition may be restarted by calling
|         'execute' again.
|
| finished(...)
|     finished( (DataAcquisitionModule)arg1) -> bool :
|         Check if the acquisition has finished. Returns True if finished.
|
| get(...)
|     get( (DataAcquisitionModule)arg1, (str)arg2, (bool)arg3) -> object :
|         Return a dict with all nodes from the specified sub-tree.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node. Use wild card to
|             select all.
|         arg3[optional]: Specify which type of data structure to return.
|             Return data either as a flat dict (True) or as a nested
|             dict tree (False). Default = False.
|
|     get( (DataAcquisitionModule)arg1, (str)arg2) -> object
|
| getDouble(...)
|     getDouble( (DataAcquisitionModule)arg1, (str)arg2) -> float :
|         Return the floating point double value for the specified path.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node.
|
| getInt(...)
|     getInt( (DataAcquisitionModule)arg1, (str)arg2) -> int :
|         Return the integer value for the specified path.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node.
|
| getString(...)
|     getString( (DataAcquisitionModule)arg1, (str)arg2) -> object :
|         Return the string value for the specified path.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node.
|
| getStringUnicode(...)
|     getStringUnicode( (DataAcquisitionModule)arg1, (str)arg2) -> object :
|         Return the unicode encoded string value for the specified path.
|         Only relevant for Python versions older than V3.0.
|         For Python versions 3.0 and later, getString can be used instead.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node.
|
| listNodes(...)
|     listNodes( (DataAcquisitionModule)arg1, (str)arg2, (int)arg3) -> list :
|         This function returns a list of node names found at the
|         specified path.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path for which the nodes should be listed. The path may
|             contain wildcards so that the returned nodes do not
|             necessarily have to have the same parents.
|         arg3: Enum that specifies how the selected nodes are listed.

```

```

        ziPython.ziListEnum.none -> 0x00
            The default flag, returning a simple
            listing if the given node
        ziPython.ziListEnum.recursive -> 0x01
            Returns the nodes recursively
        ziPython.ziListEnum.absolute -> 0x02
            Returns absolute paths
        ziPython.ziListEnum.leafsonly -> 0x04
            Returns only nodes that are leafs,
            which means the they are at the
            outermost level of the tree.
        ziPython.ziListEnum.settingsonly -> 0x08
            Returns only nodes which are marked
            as setting
        Or combinations of flags can be used.

    progress(...)
        progress( (DataAcquisitionModule)arg1) -> object :
            Reports the progress of the measurement with a number between
            0 and 1.

    read(...)
        read( (DataAcquisitionModule)arg1, (bool)arg2) -> object :
            Read acquired data. If the acquisition is still ongoing only a subset
            of the data is returned. If many triggers or huge data sets
            are acquired call this method to keep memory usage reasonable.
            arg1[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

        read( (DataAcquisitionModule)arg1) -> object

    save(...)
        save( (DataAcquisitionModule)arg1, (str)arg2) -> None :
            Save trigger data to file.
            arg1: Reference to the DataAcquisitionModule class.
            arg2: File name string (without extension).

    set(...)
        set( (DataAcquisitionModule)arg1, (str)arg2, (float)arg3) -> None :
            Data Acquisition Module Parameters
            Path name          Type      Description
            dataAcquisitionModule/buffercount
                                   int      The number of buffers used for data
acquisition.
                                   (read-only)
            dataAcquisitionModule/buffersize
                                   double   The buffersize [s] of the data
                                   acquisition object. (read-only)
            dataAcquisitionModule/flags
                                   int      Data acquisition flags.
                                   FILL = 0x0001 : Not supported by module.
                                   Grid mode will always
                                   fill data loss holes.
                                   ALIGN = 0x0002 : Not supported by module.
                                   Grid Mode will always
                                   align data.
                                   THROW = 0x0004 : Throw EOFError exception
                                   if sample loss is
                                   detected.
                                   DETECT = 0x0008: Detect data loss holes.
                                   This flag is always
                                   enabled.
            dataAcquisitionModule/device
                                   string   The device ID to execute the data
                                   acquisition, e.g. dev123
            dataAcquisitionModule/enable

```

	bool	Enable the module.
dataAcquisitionModule/endless	bool	Enable endless triggering: 1 = enable, 0 = disable.
dataAcquisitionModule/fft/absolute	bool	Shifts the frequencies so that the center frequency becomes the demodulation frequency rather than 0 Hz.
dataAcquisitionModule/fft/window	int	FFT window (default 1 = Hann) 0 = Rectangular 1 = Hann 2 = Hamming 3 = Blackman Harris 4 term 16 = Exponential (ring-down) 17 = Cosine (ring-down) 18 = Cosine squared (ring-down)
dataAcquisitionModule/forcetrigger	bool	Force a trigger.
dataAcquisitionModule/awgcontrol	bool	Enable interaction with AWG. If enabled the hwtrigger index counter will be used to control the grid row for data acquisition.
dataAcquisitionModule/triggernode	string	Path and signal of the node that should be used for triggering, separated by a dot (.), e.g. /devN/demods/0/sample.x
dataAcquisitionModule/count	int	Number of trigger edges to acquire.
dataAcquisitionModule/type	int	Trigger type used. Some parameters are only valid for special trigger nodes and/or types. 0 = trigger off 1 = analog edge trigger on source 2 = digital trigger mode on DIO 3 = analog pulse trigger on source 4 = analog tracking trigger on source 5 = change trigger 6 = hardware trigger on trigger line source. 7 = tracking edge trigger on source 8 = event count trigger on counter source.
dataAcquisitionModule/edge	int	Trigger edge 1 = rising edge 2 = falling edge 3 = both
dataAcquisitionModule/findlevel	bool	Automatically find the value of dataAcquisitionModule/level based on the current signal value.
dataAcquisitionModule/bits	int	Digital trigger condition.
dataAcquisitionModule/bitmask	int	Bit masking for bits used for triggering. Used for digital trigger.
dataAcquisitionModule/delay	double	Trigger frame position [s] (left side) relative to trigger edge. delay = 0 -> trigger edge at left border. delay < 0 -> trigger edge inside trigger frame (pretrigger).


```

|                                     delay > 0 -> trigger edge before
|                                     trigger frame (posttrigger).
|
| dataAcquisitionModule/duration
|     double Data acquisition frame length [s].
| dataAcquisitionModule/level
|     double Trigger level voltage [V].
| dataAcquisitionModule/hysteresis
|     double Trigger hysteresis [V].
| dataAcquisitionModule/triggered
|     bool    Has a data acquisition trigger been
|             triggered? 1=Yes, 0=No (read-only).
| dataAcquisitionModule/bandwidth
|     double Filter bandwidth [Hz] for pulse and
|             tracking triggers.
| dataAcquisitionModule/holdoff/count
|     int     Number of skipped triggers until the
|             next trigger is acquired again.
| dataAcquisitionModule/holdoff/time
|     double Hold off time [s] before the next
|             trigger is acquired again. A hold off
|             time smaller than the duration will
|             produce overlapped trigger frames.
| dataAcquisitionModule/pulse/min
|     double Minimum pulse width [s] for the pulse
|             trigger.
| dataAcquisitionModule/pulse/max
|     double Maximum pulse width [s] for the pulse
|             trigger.
| dataAcquisitionModule/eventcount/mode
|     int     Specifies the mode used for event count
|             processing.
|             0 - Trigger on every event count sample.
|             1 - Trigger if event count value
|                 incremented.
| dataAcquisitionModule/grid/mode
|     int     Specify how the captured data is mapped
|             onto the grid. Each trigger becomes a row
|             in the matrix and each trigger's data is
|             mapped onto a new grid row defined by the
|             number of columns using this setting:
|             1: Use nearest neighbour interpolation.
|             2: Use with linear interpolation.
|             4: Use exact alignment to the grid.
|             In this mode the duration is
|             determined from the number of grid
|             columns and the highest data sampling
|             rate of the signals to be captured.
| dataAcquisitionModule/grid/overwrite
|     bool    If enabled, the module will return only
|             one data chunk when it is running, which
|             is overwritten by consecutive triggers.
| dataAcquisitionModule/grid/cols
|     int     Specify the number of columns in the
|             grid's matrix. The data from each row is
|             mapped onto the grid according to the
|             grid/mode setting with the
|             specified number of columns.
| dataAcquisitionModule/grid/rows
|     int     Specify the number of rows in the grid's
|             matrix. Each row is the data recorded from
|             one trigger mapped onto the columns.
| dataAcquisitionModule/grid/repetitions
|     int     Number of statistical operations performed
|             per grid.
| dataAcquisitionModule/grid/rowrepetition
|     bool    If enabled, apply repetitions to row
|             instead of full grid

```

```

dataAcquisitionModule/grid/direction
    int    The direction to organize data in the
           grid's matrix:
           0: Forward.
               The data in each row is ordered chronologically, e.g., the first data point
               in each row corresponds to the first timestamp in the trigger data.
           1: Reverse.
               The data in each row is ordered reverse chronologically, e.g., the first data
               point in each row corresponds to the last timestamp in the trigger data.
           2: Bidirectional.
               The ordering of the data alternates between Forward and Backward ordering
               from row-to-row. The first row is Forward ordered.

dataAcquisitionModule/grid/waterfall
    bool    If enabled, the last grid row is always updated and the previously captured rows
           are shifted by one.

dataAcquisitionModule/refreshrate
    double  Set the rate at which the triggers are processed (Hz).

dataAcquisitionModule/save/directory
    string  The base directory where files are saved.

dataAcquisitionModule/save/filename
    string  Defines the sub-directory where files are saved. The actual sub-directory
           has this name with an incrementing sequence count (per save) appended,
           e.g. swtrigger_000.

dataAcquisitionModule/save/fileformat
    string  The format of the file for saving data:
           0 = Matlab,
           1 = CSV,
           2 = SXM (Image format).

dataAcquisitionModule/save/csvseparator
    string  The character to use as CSV separator when saving files in this format.

dataAcquisitionModule/save/csvlocale
    string  The locale to use for the decimal point character and digit grouping
           character for numerical values in CSV files:
           "C" (default) = dot for the decimal point and no digit grouping,
           "" (empty string) = use the symbols set in the language and region
           settings of the computer.

dataAcquisitionModule/save/save
    bool    Initiate the saving of data to file. The saving is done in the background.
           When the save is finished, this parameter goes low.

dataAcquisitionModule/save/saveonread
    bool    Automatically save the data to file immediately before reading the data
           out of the module using the read() command. Set this parameter when you
           want to save data to file when running the module in continuous mode
           while doing intermediate reads.

dataAcquisitionModule/historylength
    int     Sets an upper limit for the number of

```

```

|                                     data captures stored in the module.
| dataAcquisitionModule/clearhistory
|         bool    Clear all captured data from the module.
| dataAcquisitionModule/preview
|         bool    When enabled, allows the data of an
|                 incomplete trigger to be read. Useful
|                 for long data acquisitions/FFTs to
|                 display the progress.
| dataAcquisitionModule/spectrum/autobandwidth
|         bool    When set to 1, initiates automatic
|                 adjustment of the demodulator bandwidths
|                 to obtain optimal alias rejection for the
|                 selected frequency span which is
|                 equivalent to the sampling rate.
|                 The FFT mode has to be enabled
|                 (spectrum/enable) and the module has
|                 to be running for this function
|                 to take effect.
| dataAcquisitionModule/spectrum/enable
|         bool    Enables the FFT mode of the data
|                 Acquisition module, in addition to
|                 time domain.
| dataAcquisitionModule/spectrum/frequencyspan
|         double  Sets the desired frequency span of the
|                 FFT in Hz.
| dataAcquisitionModule/spectrum/overlapped
|         bool    Enables overlapping FFTs. If disabled (0),
|                 FFTs are performed on distinct abutting
|                 data sets. If enabled, the data sets of
|                 successive FFTs overlap based on the
|                 defined refresh rate.
|
| set( (DataAcquisitionModule)arg1, (str)arg2, (int)arg3) -> None
|
| set( (DataAcquisitionModule)arg1, (str)arg2, (str)arg3) -> None
|
| set( (DataAcquisitionModule)arg1, (object)arg2) -> None :
|     arg1: Reference to the DataAcquisitionModule class.
|     arg2: A list of path/value pairs.
|
| subscribe(...)
|     subscribe( (DataAcquisitionModule)arg1, (str)arg2) -> None :
|         Subscribe to one or several nodes. After subscription the acquisition
|         process can be started with the 'execute' command. During the
|         acquisition process paths can not be subscribed or unsubscribed.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node. Use wild card to
|               select all. Alternatively also a list of path
|               strings can be specified.
|
| trigger(...)
|     trigger( (DataAcquisitionModule)arg1) -> None :
|         Execute a manual trigger.
|
| unsubscribe(...)
|     unsubscribe( (DataAcquisitionModule)arg1, (str)arg2) -> None :
|         Unsubscribe from one or several nodes. During the
|         acquisition process paths can not be subscribed or unsubscribed.
|         arg1: Reference to the DataAcquisitionModule class.
|         arg2: Path string of the node. Use wild card to
|               select all. Alternatively also a list of path
|               strings can be specified.
|
| -----
| Static methods inherited from Boost.Python.instance:
|
| __new__(*args, **kwargs) from Boost.Python.class

```

```
|         Create and return a new object.  See help(type) for accurate signature.  
|  
| -----  
| Data descriptors inherited from Boost.Python.instance:  
|  
| __dict__  
|  
| __weakref__
```

5.4.7. Help for the **DeviceSettingsModule** class

An instance of DeviceSettingsModule is initialized using the deviceSettings method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.deviceSettings')
```

Help on built-in function deviceSettings in zhinst.ziPython.ziDAQServer:

```
zhinst.ziPython.ziDAQServer.deviceSettings = deviceSettings(...)
deviceSettings( (ziDAQServer)arg1) -> DeviceSettingsModule :
    Create a DeviceSettingsModule class. This will start a thread for running
    an asynchronous DeviceSettingsModule.
    arg1: Reference to the ziDAQServer class.
    arg2: Timeout in [ms]. Recommended value is 500ms.
    DEPRECATED, ignored

deviceSettings( (ziDAQServer)arg1, (int)arg2) -> DeviceSettingsModule
```

Reference help for the DeviceSettingsModule class.

```
>>> help('zhinst.ziPython.DeviceSettingsModule')
```

Help on class DeviceSettingsModule in zhinst.ziPython:

```
zhinst.ziPython.DeviceSettingsModule = class
DeviceSettingsModule(Boost.Python.instance)
|   Method resolution order:
|       DeviceSettingsModule
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       Raises an exception
|       This class cannot be instantiated from Python
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   clear(...)
|       clear( (DeviceSettingsModule)arg1) -> None :
|           End the deviceSettings thread.
|
|   execute(...)
|       execute( (DeviceSettingsModule)arg1) -> None :
|           Execute the save/load command.
|
|   finish(...)
|       finish( (DeviceSettingsModule)arg1) -> None :
|           Stop the load/save command. The command may be restarted by calling
|           'execute' again.
|
|   finished(...)
|       finished( (DeviceSettingsModule)arg1) -> bool :
```

```

|         Check if the command execution has finished. Returns True if finished.
|
| get(...)
|     get( (DeviceSettingsModule)arg1, (str)arg2, (bool)arg3) -> object :
|         Return a dict with all nodes from the specified sub-tree.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path string of the node. Use wild card to
|             select all.
|         arg3[optional]: Specify which type of data structure to return.
|             Return data either as a flat dict (True) or as a nested
|             dict tree (False). Default = False.
|
|     get( (DeviceSettingsModule)arg1, (str)arg2) -> object
|
| getDouble(...)
|     getDouble( (DeviceSettingsModule)arg1, (str)arg2) -> float :
|         Return the floating point double value for the specified path.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path string of the node.
|
| getInt(...)
|     getInt( (DeviceSettingsModule)arg1, (str)arg2) -> int :
|         Return the integer value for the specified path.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path string of the node.
|
| getString(...)
|     getString( (DeviceSettingsModule)arg1, (str)arg2) -> object :
|         Return the string value for the specified path.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path string of the node.
|
| getStringUnicode(...)
|     getStringUnicode( (DeviceSettingsModule)arg1, (str)arg2) -> object :
|         Get a unicode string value from the specified node.
|         The returned string is unicode encoded.
|         Only relevant for Python versions older than V3.0.
|         For Python versions 3.0 and later, getString can be used instead.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path string of the node.
|
| listNodes(...)
|     listNodes( (DeviceSettingsModule)arg1, (str)arg2, (int)arg3) -> list :
|         This function returns a list of node names found at the specified path.
|         arg1: Reference to the DeviceSettingsModule class.
|         arg2: Path for which the nodes should be listed. The path may
|             contain wildcards so that the returned nodes do not
|             necessarily have to have the same parents.
|         arg3: Enum that specifies how the selected nodes are listed.
|             ziPython.ziListEnum.none -> 0x00
|                 The default flag, returning a simple
|                 listing if the given node
|             ziPython.ziListEnum.recursive -> 0x01
|                 Returns the nodes recursively
|             ziPython.ziListEnum.absolute -> 0x02
|                 Returns absolute paths
|             ziPython.ziListEnum.leafsonly -> 0x04
|                 Returns only nodes that are leafs,
|                 which means the they are at the
|                 outermost level of the tree.
|             ziPython.ziListEnum.settingsonly -> 0x08
|                 Returns only nodes which are marked
|                 as setting
|         Or combinations of flags can be used.
|
| progress(...)
|     progress( (DeviceSettingsModule)arg1) -> object :

```

```

        Reports the progress of the command with a number between
        0 and 1.

read(...)
    read( (DeviceSettingsModule)arg1, (bool)arg2) -> object :
        Read device settings. Only relevant for the save command.
        arg1[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

    read( (DeviceSettingsModule)arg1) -> object

save(...)
    save( (DeviceSettingsModule)arg1, (str)arg2) -> None :
        Not relevant for the deviceSettings module.

set(...)
    set( (DeviceSettingsModule)arg1, (str)arg2, (float)arg3) -> None :
        Device Settings Parameters
        Path name      Type      Description
        deviceSettings/device    string    Comma separated list of devices that
                                         should be used for loading/saving
                                         device settings, e.g. 'dev99,dev100'.
        deviceSettings/path      string    Directory where settings files should be
                                         located. If not set, the default settings
                                         location of the LabOne software is used.
        deviceSettings/filename  string    Name of settings file to use
        deviceSettings/command   string    The command to execute
                                         'save' = Read device settings and save to
                                         file.
                                         'load' = Load settings from file and
                                         write to device.
                                         'read' = Read device settings only
                                         (no save).

    set( (DeviceSettingsModule)arg1, (str)arg2, (int)arg3) -> None

    set( (DeviceSettingsModule)arg1, (str)arg2, (str)arg3) -> None

    set( (DeviceSettingsModule)arg1, (object)arg2) -> None :
        arg1: Reference to the DeviceSettingsModule class.
        arg2: A list of path/value pairs.

subscribe(...)
    subscribe( (DeviceSettingsModule)arg1, (str)arg2) -> None :
        Not relevant for the deviceSettings module.

trigger(...)
    trigger( (DeviceSettingsModule)arg1) -> None :
        Not applicable to this module.

unsubscribe(...)
    unsubscribe( (DeviceSettingsModule)arg1, (str)arg2) -> None :
        Not relevant for the deviceSettings module.

-----
Static methods inherited from Boost.Python.instance:

__new__(*args, **kwargs) from Boost.Python.class
    Create and return a new object. See help(type) for accurate signature.

-----
Data descriptors inherited from Boost.Python.instance:

__dict__
__weakref__

```

5.4.8. Help for the **ImpedanceModule** class

An instance of `ImpedanceModule` is initialized using the `impedanceModule` method from `ziDAQServer`:

```
>>> help('zhinst.ziPython.ziDAQServer.impedanceModule')
```

Help on built-in function `impedanceModule` in `zhinst.ziPython.ziDAQServer`:

```
zhinst.ziPython.ziDAQServer.impedanceModule = impedanceModule(...)
  impedanceModule( (ziDAQServer)arg1) -> ImpedanceModule :
    Create a ImpedanceModule class. This will start a thread for
    running an asynchronous ImpedanceModule.
    arg1: Reference to the ziDAQServer class.
```

Reference help for the `ImpedanceModule` class.

```
>>> help('zhinst.ziPython.ImpedanceModule')
```

Help on class `ImpedanceModule` in `zhinst.ziPython`:

```
zhinst.ziPython.ImpedanceModule = class ImpedanceModule(Boost.Python.instance)
| Method resolution order:
|   ImpedanceModule
|   Boost.Python.instance
|   builtins.object
|
| Static methods defined here:
|
|   __init__(...)
|       Raises an exception
|       This class cannot be instantiated from Python
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   clear(...)
|       clear( (ImpedanceModule)arg1) -> None :
|           End the ImpedanceModule thread.
|
|   execute(...)
|       execute( (ImpedanceModule)arg1) -> None :
|           Starts the ImpedanceModule if not yet running.
|
|   finish(...)
|       finish( (ImpedanceModule)arg1) -> None :
|           Stop the ImpedanceModule.
|
|   finished(...)
|       finished( (ImpedanceModule)arg1) -> bool :
|           Check if the command execution has finished. Returns True if finished.
|
|   get(...)
|       get( (ImpedanceModule)arg1, (str)arg2, (bool)arg3) -> object :
|           Return a dict with all nodes from the specified sub-tree.
|           arg1: Reference to the ImpedanceModule class.
|           arg2: Path string of the node. Use wild card to
|               select all.
|           arg3[optional]: Specify which type of data structure to return.
|               Return data either as a flat dict (True) or as a nested
|               dict tree (False). Default = False.
|
|       get( (ImpedanceModule)arg1, (str)arg2) -> object
```

```

| getDouble(...)
|     getDouble( (ImpedanceModule)arg1, (str)arg2) -> float :
|         Return the floating point double value for the specified path.
|         arg1: Reference to the ImpedanceModule class.
|         arg2: Path string of the node.
|
| getInt(...)
|     getInt( (ImpedanceModule)arg1, (str)arg2) -> int :
|         Return the integer value for the specified path.
|         arg1: Reference to the ImpedanceModule class.
|         arg2: Path string of the node.
|
| getString(...)
|     getString( (ImpedanceModule)arg1, (str)arg2) -> object :
|         Return the string value for the specified path.
|         arg1: Reference to the ImpedanceModule class.
|         arg2: Path string of the node.
|
| getStringUnicode(...)
|     getStringUnicode( (ImpedanceModule)arg1, (str)arg2) -> object :
|         Get a unicode string value from the specified node.
|         The returned string is unicode encoded.
|         Only relevant for Python versions older than V3.0.
|         For Python versions 3.0 and later, getString can be used instead.
|         arg1: Reference to the ImpedanceModule class.
|         arg2: Path string of the node.
|
| listNodes(...)
|     listNodes( (ImpedanceModule)arg1, (str)arg2, (int)arg3) -> list :
|         This function returns a list of node names found at the specified path.
|         arg1: Reference to the ImpedanceModule class.
|         arg2: Path for which the nodes should be listed. The path may
|             contain wildcards so that the returned nodes do not
|             necessarily have to have the same parents.
|         arg3: Enum that specifies how the selected nodes are listed.
|             ziPython.ziListEnum.none -> 0x00
|                 The default flag, returning a simple
|                 listing if the given node
|             ziPython.ziListEnum.recursive -> 0x01
|                 Returns the nodes recursively
|             ziPython.ziListEnum.absolute -> 0x02
|                 Returns absolute paths
|             ziPython.ziListEnum.leafsonly -> 0x04
|                 Returns only nodes that are leafs,
|                 which means the they are at the
|                 outermost level of the tree.
|             ziPython.ziListEnum.settingsonly -> 0x08
|                 Returns only nodes which are marked
|                 as setting
|         Or combinations of flags can be used.
|
| progress(...)
|     progress( (ImpedanceModule)arg1) -> object :
|         Reports the progress of the command with a number between 0 and 1.
|
| save(...)
|     save( (ImpedanceModule)arg1, (str)arg2) -> None :
|         Not relevant for the ImpedanceModule.
|
| set(...)
|     set( (ImpedanceModule)arg1, (str)arg2, (float)arg3) -> None :
|         Impedance Module Parameters
|         Path name      Type      Description
|         impedanceModule/directory string The directory where files are saved.
|         impedanceModule/calibrate bool   If set to true will execute a
|                                           compensation for the specified
|                                           compensation condition.

```


impedanceModule/device	string	Device string defining the device on which the compensation is performed.
impedanceModule/step	int	Compensation step to be performed when calibrate indicator is set to true. 0 - First load 1 - Second load 2 - Third load 3 - Fourth load
impedanceModule/mode	int	Compensation mode to be used. Defines which load steps need to be compensated. 3 - SO (Short-Open) 4 - L (Load) 5 - SL (Short-Load) 6 - OL (Open-Load) 7 - SOL (Short-Open-Load) 8 - LLL (Load-Load-Load)
impedanceModule/status	int	Bit coded field of the already compensated load conditions (bit 0 = first load).
impedanceModule/loads/0/r	double	Resistance value of first compensation load (SHORT) [Ohm]
impedanceModule/loads/1/r	double	Resistance value of second compensation load (OPEN) [Ohm]
impedanceModule/loads/2/r	double	Resistance value of third compensation load (LOAD) [Ohm]
impedanceModule/loads/3/r	double	Resistance value of the fourth compensation load (LOAD) [Ohm]. This load setting is only used if high impedance load is enabled.
impedanceModule/loads/0/c	double	Parallel capacitance of the first compensation load (SHORT) [F]
impedanceModule/loads/1/c	double	Parallel capacitance of the second compensation load (OPEN) [F]
impedanceModule/loads/2/c	double	Parallel capacitance of the third compensation load (LOAD) [F]
impedanceModule/loads/3/c	double	Parallel capacitance of the fourth compensation load (LOAD) [F]
impedanceModule/freq/start	double	Start frequency of compensation traces [Hz]
impedanceModule/freq/stop	double	Stop frequency of compensation traces [Hz]
impedanceModule/freq/samplecount	int	Number of samples of a compensation trace
impedanceModule/highimpedanceload	bool	Enable a second high impedance load compensation for the low current ranges.
impedanceModule/expectedstatus	int	Bit field of the load condition that the corresponds a full compensation. If status is equal the expected status the compensation is complete.
impedanceModule/message	string	Message string containing information, warnings or error messages during compensation.
impedanceModule/comment	string	Comment string that will be saved together with the compensation data.
impedanceModule/validation	bool	Enable the validation of compensation data. If enabled the compensation is checked for too big deviation from specified load.
impedanceModule/precision	int	Precision of the compensation. Will affect time of a compensation and reduces the noise on compensation traces.

```

|                                     0 - Standard speed
|                                     1 - Low speed / high precision
| impedanceModule/todevice    bool   If enabled will automatically
|                                     transfer compensation data to the
|                                     persistent flash memory in case of a
|                                     valid compensation.
| impedanceModule/progress    double  Progress of a compensation condition.
|
| set( (ImpedanceModule)arg1, (str)arg2, (int)arg3) -> None
|
| set( (ImpedanceModule)arg1, (str)arg2, (str)arg3) -> None
|
| set( (ImpedanceModule)arg1, (object)arg2) -> None :
|     arg1: Reference to the ImpedanceModule class.
|     arg2: A list of path/value pairs.
|
| subscribe(...)
|     subscribe( (ImpedanceModule)arg1, (str)arg2) -> None :
|         Not relevant for the ImpedanceModule.
|
| trigger(...)
|     trigger( (ImpedanceModule)arg1) -> None :
|         Not applicable to this module.
|
| unsubscribe(...)
|     unsubscribe( (ImpedanceModule)arg1, (str)arg2) -> None :
|         Not relevant for the ImpedanceModule.
|
| -----
| Static methods inherited from Boost.Python.instance:
|
| __new__(*args, **kwargs) from Boost.Python.class
|     Create and return a new object. See help(type) for accurate signature.
|
| -----
| Data descriptors inherited from Boost.Python.instance:
|
| __dict__
|
| __weakref__

```

5.4.9. Help for the **MultiDeviceSyncModule** class

An instance of `MultiDeviceSyncModule` is initialized using the `multiDeviceSyncModule` method from `ziDAQServer`:

```

>>> help('zhinst.ziPython.ziDAQServer.multiDeviceSyncModule')

Help on built-in function multiDeviceSyncModule in zhinst.ziPython.ziDAQServer:

zhinst.ziPython.ziDAQServer.multiDeviceSyncModule = multiDeviceSyncModule(...)
multiDeviceSyncModule( (ziDAQServer)arg1) -> MultiDeviceSyncModule :
    Create a MultiDeviceSyncModule class. This will start a thread for
    running an asynchronous MultiDeviceSync module.
    arg1: Reference to the ziDAQServer class.

```

Reference help for the `MultiDeviceSyncModule` class.

```

>>> help('zhinst.ziPython.MultiDeviceSyncModule')

Help on class MultiDeviceSyncModule in zhinst.ziPython:

zhinst.ziPython.MultiDeviceSyncModule = class
MultiDeviceSyncModule(Boost.Python.instance)

```

```

| Method resolution order:
|     MultiDeviceSyncModule
|     Boost.Python.instance
|     builtins.object
|
| Static methods defined here:
|
| __init__(...)
|     Raises an exception
|     This class cannot be instantiated from Python
|
| __reduce__ = <unnamed Boost.Python function>(...)
|
| clear(...)
|     clear( (MultiDeviceSyncModule)arg1) -> None :
|         End the MultiDeviceSyncModule thread.
|
| execute(...)
|     execute( (MultiDeviceSyncModule)arg1) -> None :
|         Starts the MultiDeviceSyncModule if not yet running.
|
| finish(...)
|     finish( (MultiDeviceSyncModule)arg1) -> None :
|         Stop the MultiDeviceSync module.
|
| finished(...)
|     finished( (MultiDeviceSyncModule)arg1) -> bool :
|         Check if the command execution has finished. Returns True if finished.
|
| get(...)
|     get( (MultiDeviceSyncModule)arg1, (str)arg2, (bool)arg3) -> object :
|         Return a dict with all nodes from the specified sub-tree.
|         arg1: Reference to the MultiDeviceSyncModule class.
|         arg2: Path string of the node. Use a wildcard to
|             select all.
|         arg3[optional]: Specify which type of data structure to return.
|             Return data either as a flat dict (True) or as a nested
|             dict tree (False). Default = False.
|
|     get( (MultiDeviceSyncModule)arg1, (str)arg2) -> object
|
| getDouble(...)
|     getDouble( (MultiDeviceSyncModule)arg1, (str)arg2) -> float :
|         Return the floating point double value for the specified path.
|         arg1: Reference to the MultiDeviceSyncModule class.
|         arg2: Path string of the node.
|
| getInt(...)
|     getInt( (MultiDeviceSyncModule)arg1, (str)arg2) -> int :
|         Return the integer value for the specified path.
|         arg1: Reference to the MultiDeviceSyncModule class.
|         arg2: Path string of the node.
|
| getString(...)
|     getString( (MultiDeviceSyncModule)arg1, (str)arg2) -> object :
|         Return the string value for the specified path.
|         arg1: Reference to the MultiDeviceSyncModule class.
|         arg2: Path string of the node.
|
| getStringUnicode(...)
|     getStringUnicode( (MultiDeviceSyncModule)arg1, (str)arg2) -> object :
|         Get a unicode string value from the specified node.
|         The returned string is unicode encoded.
|         Only relevant for Python versions older than V3.0.
|         For Python versions 3.0 and later, getString can be used instead.
|         arg1: Reference to the MultiDeviceSyncModule class.
|         arg2: Path string of the node.

```

```

listNodes(...)
    listNodes( (MultiDeviceSyncModule)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the MultiDeviceSyncModule class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            ziPython.ziListEnum.none -> 0x00
                The default flag, returning a simple
                listing if the given node
            ziPython.ziListEnum.recursive -> 0x01
                Returns the nodes recursively
            ziPython.ziListEnum.absolute -> 0x02
                Returns absolute paths
            ziPython.ziListEnum.leafsonly -> 0x04
                Returns only nodes that are leafs,
                which means the they are at the
                outermost level of the tree.
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
            Or combinations of flags can be used.

progress(...)
    progress( (MultiDeviceSyncModule)arg1) -> object :
        Reports the progress of the command with a number between 0 and 1.

save(...)
    save( (MultiDeviceSyncModule)arg1, (str)arg2) -> None :
        Not relevant for the multiDeviceSyncModule.

set(...)
    set( (MultiDeviceSyncModule)arg1, (str)arg2, (float)arg3) -> None :
        MultiDeviceSyncModule Parameters
        Path name                                Type    Description
        multiDeviceSyncModule/devices            string  Defines which instruments should
                                                be included in the
                                                synchronization. Expects a comma-
                                                separated list of devices in the
                                                order the devices are connected.
        multiDeviceSyncModule/group              int     Defines in which synchronization
                                                group should be accessed by the
                                                module.
        multiDeviceSyncModule/message            int     Status message of the module.
        multiDeviceSyncModule/start              bool    Set to true to start the
                                                synchronization process.
        multiDeviceSyncModule/status             int     Status of the synchronization
                                                process.
                                                -1 - error
                                                0 - idle
                                                1 - synchronization in
                                                  progress
                                                2 - synchronization successful
        multiDeviceSyncModule/recover            bool    Set to enable recovery of already
                                                synchronized devices.
                                                0 - off (default)
                                                1 - on
        multiDeviceSyncModule/phasesync          bool    Set to 1 to start phase reset on
                                                alloscillators of synches
                                                devices.

    set( (MultiDeviceSyncModule)arg1, (str)arg2, (int)arg3) -> None

    set( (MultiDeviceSyncModule)arg1, (str)arg2, (str)arg3) -> None

```

```
|         set( (MultiDeviceSyncModule)arg1, (object)arg2) -> None :  
|             arg1: Reference to the MultiDeviceSyncModule class.  
|             arg2: A list of path/value pairs.  
|  
|         subscribe(...)  
|             subscribe( (MultiDeviceSyncModule)arg1, (str)arg2) -> None :  
|                 Not relevant for the multiDeviceSyncModule.  
|  
|         trigger(...)  
|             trigger( (MultiDeviceSyncModule)arg1) -> None :  
|                 Not applicable to this module.  
|  
|         unsubscribe(...)  
|             unsubscribe( (MultiDeviceSyncModule)arg1, (str)arg2) -> None :  
|                 Not relevant for the multiDeviceSyncModule.  
|  
| -----  
| Static methods inherited from Boost.Python.instance:  
|  
| __new__(*args, **kwargs) from Boost.Python.class  
|     Create and return a new object.  See help(type) for accurate signature.  
|  
| -----  
| Data descriptors inherited from Boost.Python.instance:  
|  
| __dict__  
|  
| __weakref__
```

5.4.10. Help for the **PidAdvisorModule** class

An instance of `PidAdvisorModule` is initialized using the `pidAdvisor` method from `ziDAQServer`:

```
>>> help('zhinst.ziPython.ziDAQServer.pidAdvisor')
```

Help on built-in function `pidAdvisor` in `zhinst.ziPython.ziDAQServer`:

```
zhinst.ziPython.ziDAQServer.pidAdvisor = pidAdvisor(...)
pidAdvisor( (ziDAQServer)arg1) -> PidAdvisorModule :
    Create a PidAdvisorModule class. This will start a thread for running an
    asynchronous PidAdvisorModule.
    arg1: Reference to the ziDAQServer class.
    arg2: Timeout in [ms]. Recommended value is 500ms.
    DEPRECATED, ignored

pidAdvisor( (ziDAQServer)arg1, (int)arg2) -> PidAdvisorModule
```

Reference help for the `PidAdvisorModule` class.

```
>>> help('zhinst.ziPython.PidAdvisorModule')
```

Help on class `PidAdvisorModule` in `zhinst.ziPython`:

```
zhinst.ziPython.PidAdvisorModule = class PidAdvisorModule(Boost.Python.instance)
|   Method resolution order:
|       PidAdvisorModule
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       Raises an exception
```

```

    This class cannot be instantiated from Python

__reduce__ = <unnamed Boost.Python function>(...)

clear(...)
    clear( (PidAdvisorModule)arg1) -> None :
        End the pidAdvisor thread.

execute(...)
    execute( (PidAdvisorModule)arg1) -> None :
        Starts the pidAdvisor if not yet running.

finish(...)
    finish( (PidAdvisorModule)arg1) -> None :
        Stop the pidAdvisor.

finished(...)
    finished( (PidAdvisorModule)arg1) -> bool :
        Check if the command execution has finished. Returns True if finished.

get(...)
    get( (PidAdvisorModule)arg1, (str)arg2, (bool)arg3) -> object :
        Return a dict with all nodes from the specified sub-tree.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path string of the node. Use wild card to
            select all.
        arg3[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

    get( (PidAdvisorModule)arg1, (str)arg2) -> object

getDouble(...)
    getDouble( (PidAdvisorModule)arg1, (str)arg2) -> float :
        Return the floating point double value for the specified path.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path string of the node.

getInt(...)
    getInt( (PidAdvisorModule)arg1, (str)arg2) -> int :
        Return the integer value for the specified path.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path string of the node.

getString(...)
    getString( (PidAdvisorModule)arg1, (str)arg2) -> object :
        Return the string value for the specified path.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path string of the node.

getStringUnicode(...)
    getStringUnicode( (PidAdvisorModule)arg1, (str)arg2) -> object :
        Get a unicode string value from the specified node.
        The returned string is unicode encoded.
        Only relevant for Python versions older than V3.0.
        For Python versions 3.0 and later, getString can be used instead.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path string of the node.

listNodes(...)
    listNodes( (PidAdvisorModule)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the PidAdvisorModule class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.

```

```

        ziPython.ziListEnum.none -> 0x00
            The default flag, returning a simple
            listing of the given node
        ziPython.ziListEnum.recursive -> 0x01
            Returns the nodes recursively
        ziPython.ziListEnum.absolute -> 0x02
            Returns absolute paths
        ziPython.ziListEnum.leafsonly -> 0x04
            Returns only leaf nodes,
            which means the they are at the
            outermost level of the tree.
        ziPython.ziListEnum.settingsonly -> 0x08
            Returns only nodes which are marked
            as settings
        Or combinations of flags can be used.

progress(...)
    progress( (PidAdvisorModule)arg1) -> object :
        Reports the progress of the command with a number between 0 and 1.

read(...)
    read( (PidAdvisorModule)arg1, (bool)arg2) -> object :
        Read pidAdvisor data. If the simulation is still ongoing, only a subset
        of the data is returned.
        arg1[optional]: Specify which type of data structure to return.
        Return data either as a flat dict (True) or as a nested
        dict tree (False). Default = False.

    read( (PidAdvisorModule)arg1) -> object

save(...)
    save( (PidAdvisorModule)arg1, (str)arg2) -> None :
        Save PID advisor data to file.
        arg1: Reference to the PidAdvisorModule class.
        arg2: File name string (without extension).

set(...)
    set( (PidAdvisorModule)arg1, (str)arg2, (float)arg3) -> None :
        PID Advisor Parameters
        Path name      Type      Description
        pidAdvisor/advancedmode    int      Disable automatic calculation of
                                         the start and stop value.
        pidAdvisor/auto            int      Automatic response calculation
                                         triggered by parameter change.
        pidAdvisor/bode            struct    Output parameter. Contains the
                                         resulting bode plot of the PID
                                         simulation.
        pidAdvisor/bw              double    Output parameter. Calculated system
                                         bandwidth.
        pidAdvisor/calculate       int      In/Out parameter. Command to
                                         calculate values. Set to 1 to start
                                         the calculation.
        pidAdvisorordemod/timeconstant double    Demodulator filter time constant to
                                         be used for the advisor.
        pidAdvisorordemod/order    double    Demodulator filter order for the
                                         advisor.
        pidAdvisorordemod/harmonic double    Demodulator harmonic used by the
                                         advisor.
        pidAdvisor/display/freqstart double    Start frequency for Bode plot.
                                         For disabled advanced mode the
                                         start value is automatically
                                         derived from the system properties.
        pidAdvisor/display/freqstop double    Stop frequency for Bode plot.
        pidAdvisor/display/timestart double    Start time for step response.
        pidAdvisor/display/timestop double    Stop time for step response.
        pidAdvisor/dut/bw          double    Bandwidth of the DUT (device under
                                         test).

```

pidAdvisor/dut/damping	double	Damping of the second order low pass filter.
pidAdvisor/dut/delay	double	IO Delay of the feedback system describing the earliest response for a step change.
pidAdvisor/dut/fcenter	double	Resonant frequency of the of the modelled resonator.
pidAdvisor/dut/gain	double	Gain of the DUT transfer function.
pidAdvisor/dut/q	double	quality factor of the modelled resonator.
pidAdvisor/dut/source	int	Type of model used for the external device to be controlled by the PID. source = 1: Low-pass first order source = 2: Low-pass second order source = 3: Resonator frequency source = 4: Internal PLL source = 5: VCO source = 6: Resonator amplitude
pidAdvisor/impulse	struct	Output parameter. Impulse response (not yet supported).
pidAdvisor/index	int	PID index for parameter detection.
pidAdvisor/pid/autobw	int	Adjusts the demodulator bandwidth to fit best to the specified target bandwidth of the full system.
pidAdvisor/pid/autolimit	int	Switch the writing of PID limits when 'To PID' is pressed. Only applies in case of internal PLL.
pidAdvisor/pid/d	double	In/Out parameter. Differential gain.
pidAdvisor/pid/dlimittimeconstant	double	In/Out parameter. Differential filter timeconstant.
pidAdvisor/pid/i	double	In/Out parameter. Integral gain.
pidAdvisor/pid/mode	double	Select PID Advisor mode. Mode value is bit coded, bit 0: P, bit 1: I, bit 2: D, bit 3: D filter limit.
pidAdvisor/pid/p	double	In/Out parameter. Proportional gain.
pidAdvisor/pid/rate	double	In/Out parameter. PID Advisor sampling rate of the PID control loop.
pidAdvisor/pid/targetbw	double	PID system target bandwidth.
pidAdvisor/todevice	int	Set to 1 to transfer PID advisor data to the device.
pidAdvisor/type	string	If set to 'pll' for an HF2 device the PID advisor will advise parameters for the hardware PLL. The default is 'pid'. This parameter is only relevant for. HF2 devices.
pidAdvisor/pm	double	Output parameter. Simulated phase margin of the PID with the current settings. The phase margin should be greater than 45 deg and preferably greater than 65 deg for stable conditions.
pidAdvisor/pmfreq	double	Output parameter. Simulated phase margin frequency.
pidAdvisor/response	int	In/Out parameter. Command to calculate the advisor response. Set to 1 to start
pidAdvisor/stable	int	Output parameter. When 1, the PID Advisor found a stable solution with the given settings. When 0, revise your settings and rerun the PID Advisor.
pidAdvisor/step	struct	Output parameter. Contains the

		resulting step response plot of the PID simulation.
pidAdvisor/targetbw	double	Requested PID bandwidth. Higher frequencies may need manual tuning.
pidAdvisor/targetfail	int	Output parameter. 1 indicates the simulated PID BW is smaller than the Target BW.
pidAdvisor/tf/closedloop	int	Switch the response calculation mode between closed or open loop.
pidAdvisor/tf/input	int	Start point for the plant response simulation for open or closed loops.
pidAdvisor/tf/output	int	End point for the plant response simulation for open or closed loops.
pidAdvisor/tune	int	Optimize the PID parameters so that the noise of the closed-loop system gets minimized. The HF2 doesn't support tuning.
pidAdvisor/tuner/mode	int	Select tuner mode. Mode value is bit coded, bit 0: P, bit 1: I, bit 2: D, bit 3: D filter limit.
pidAdvisor/tuner/averagetime	double	Time for a tuner iteration.
set((PidAdvisorModule)arg1, (str)arg2, (int)arg3) -> None		
set((PidAdvisorModule)arg1, (str)arg2, (str)arg3) -> None		
set((PidAdvisorModule)arg1, (object)arg2) -> None :		
arg1: Reference to the PidAdvisor class.		
arg2: A list of path/value pairs.		
subscribe(...)		
subscribe((PidAdvisorModule)arg1, (str)arg2) -> None :		
Subscribe to one or several nodes.		
trigger(...)		
trigger((PidAdvisorModule)arg1) -> None :		
Not applicable to this module.		
unsubscribe(...)		
unsubscribe((PidAdvisorModule)arg1, (str)arg2) -> None :		
Unsubscribe from one or several nodes.		

Static methods inherited from Boost.Python.instance:		
__new__(*args, **kwargs) from Boost.Python.class		
Create and return a new object. See help(type) for accurate signature.		

Data descriptors inherited from Boost.Python.instance:		
__dict__		
__weakref__		

5.4.11. Help for the **ScopeModule** class

An instance of ScopeModule is initialized using the scopeModule method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.scopeModule')
```

Help on built-in function scopeModule in zhinst.ziPython.ziDAQServer:

```
zhinst.ziPython.ziDAQServer.scopeModule = scopeModule(...)
scopeModule( (ziDAQServer)arg1) -> ScopeModule :
    Create a ScopeModule class. This will start a thread for running an
    asynchronous ScopeModule.
    arg1: Reference to the ziDAQServer class.
```

Reference help for the ScopeModule class.

```
>>> help('zhinst.ziPython.ScopeModule')
```

Help on class ScopeModule in zhinst.ziPython:

```
zhinst.ziPython.ScopeModule = class ScopeModule(Boost.Python.instance)
| Method resolution order:
|     ScopeModule
|     Boost.Python.instance
|     builtins.object
|
| Static methods defined here:
|
|     __init__(...)
|         Raises an exception
|         This class cannot be instantiated from Python
|
|     __reduce__ = <unnamed Boost.Python function>(...)
|
|     clear(...)
|         clear( (ScopeModule)arg1) -> None :
|             End the ScopeModule thread.
|
|     execute(...)
|         execute( (ScopeModule)arg1) -> None :
|             Starts the ScopeModule if not yet running.
|
|     finish(...)
|         finish( (ScopeModule)arg1) -> None :
|             Stop the ScopeModule module.
|
|     finished(...)
|         finished( (ScopeModule)arg1) -> bool :
|             Scope module is always running, use progress instead
|
|     get(...)
|         get( (ScopeModule)arg1, (str)arg2, (bool)arg3) -> object :
|             Return a dict with all nodes from the specified sub-tree.
|             arg1: Reference to the ScopeModule class.
|             arg2: Path string of the node. Use a wildcard to
|                 select all.
|             arg3[optional]: Specify which type of data structure to return.
|                 Return data either as a flat dict (True) or as a nested
|                 dict tree (False). Default = False.
|
|         get( (ScopeModule)arg1, (str)arg2) -> object
|
|     getDouble(...)
|         getDouble( (ScopeModule)arg1, (str)arg2) -> float :
|             Return the floating point double value for the specified path.
|             arg1: Reference to the ScopeModule class.
|             arg2: Path string of the node.
|
|     getInt(...)
|         getInt( (ScopeModule)arg1, (str)arg2) -> int :
|             Return the integer value for the specified path.
|             arg1: Reference to the ScopeModule class.
|             arg2: Path string of the node.
```

```

getString(...)
    getString( (ScopeModule)arg1, (str)arg2) -> object :
        Return the string value for the specified path.
        arg1: Reference to the ScopeModule class.
        arg2: Path string of the node.

getStringUnicode(...)
    getStringUnicode( (ScopeModule)arg1, (str)arg2) -> object :
        Get a unicode string value from the specified node.
        The returned string is unicode encoded.
        Only relevant for Python versions older than V3.0.
        For Python versions 3.0 and later, getString can be used instead.
        arg1: Reference to the ScopeModule class.
        arg2: Path string of the node.

listNodes(...)
    listNodes( (ScopeModule)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the ScopeModule class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            ziPython.ziListEnum.none -> 0x00
                The default flag, returning a simple
                listing if the given node
            ziPython.ziListEnum.recursive -> 0x01
                Returns the nodes recursively
            ziPython.ziListEnum.absolute -> 0x02
                Returns absolute paths
            ziPython.ziListEnum.leafsonly -> 0x04
                Returns only nodes that are leafs,
                which means the they are at the
                outermost level of the tree.
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
            Or combinations of flags can be used.

progress(...)
    progress( (ScopeModule)arg1) -> object :
        Reports the progress of the command with a number between 0 and 1.

read(...)
    read( (ScopeModule)arg1, (bool)arg2) -> object :
        Read scope data. If the recording is still ongoing only a subset
        of data is returned.
        arg1[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

    read( (ScopeModule)arg1) -> object

save(...)
    save( (ScopeModule)arg1, (str)arg2) -> None :
        Save scope data to file.
        arg1: Reference to the ScopeModule class.
        arg2: File name string (without extension).

set(...)
    set( (ScopeModule)arg1, (str)arg2, (float)arg3) -> None :
        Scope module parameters
        Path name          Type      Description
        scopeModule/historylength  int      Sets an upper limit for the number
                                                of data captures stored in
                                                the module.

```

scopeModule/clearhistory	bool	Clear all captured data from the module.
scopeModule/externalscaling	double	Scaling to apply to the scope data transferred over API level 1 connection (HF2).
scopeModule/mode	int	Scope data processing mode: 0 - Pass through: scope segments assembled and returned unprocessed, non-interleaved. 1 - Moving average: entire scope recording assembled, scaling applied, averager if enabled (see weight), data returned in float non-interleaved format. 2 - Average: not implemented yet. 3 - FFT: same as mode 1, except FFT applied to every segment of the scope recording. See fft/* nodes for FFT parameters.
scopeModule/averager/weight	int	=1 - averager disabled. >1 - moving average, updating last history entry.
scopeModule/averager/restart	bool	1 - resets the averager. Action node switches back to 0. automatically
scopeModule/averager/resamplingmode	int	When averaging low sampling rate data aligned by high resolution trigger, scope data must be resampled to keep same samples positioning relative to the trigger between averaged recordings. Resample using: 0 - Linear interpolation 1 - PCHIP interpolation
scopeModule/fft/window	int	FFT Window: 0 - Rectangular 1 - Hann 2 - Hamming 3 - Blackman Harris 16 - Exponential (ring-down) 17 - Cosine (ring-down) 18 - Cosine squared (ring-down)
scopeModule/fft/power	bool	1 - Calculate power value
scopeModule/fft/spectraldensity	bool	1 - Calculate spectral density value
scopeModule/save/directory	string	The base directory where files are saved.
scopeModule/save/filename	string	Defines the sub-directory where files are saved. The actual sub-directory has this name with an incrementing sequence count (per save) appended, e.g. scope_000.
scopeModule/save/fileformat	string	The format of the file for saving data: 0 = Matlab, 1 = CSV, 2 = ZView (Impedance data only).
scopeModule/save/csvseparator	string	The character to use as CSV separator when saving files in this format.
scopeModule/save/csvlocale	string	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C" (default) = dot for the

```

decimal point and no digit grouping,
"" (empty string) = use the
symbols set in the language and
region settings of the computer.
scopeModule/save/save          bool    Initiate the saving of data to file.
                                     The saving is done in
                                     the background.
                                     When the save is finished,
                                     this parameter goes low.
scopeModule/save/saveonread    bool    Automatically save the data to file
                                     immediately before reading the data
                                     out of the module using the read()
                                     command. Set this parameter when you
                                     want to save data to file when
                                     running the module in continous mode
                                     while doing intermediate reads.

set( (ScopeModule)arg1, (str)arg2, (int)arg3) -> None

set( (ScopeModule)arg1, (str)arg2, (str)arg3) -> None

set( (ScopeModule)arg1, (object)arg2) -> None :
    arg1: Reference to the ScopeModule class.
    arg2: A list of path/value pairs.

subscribe(...)
    subscribe( (ScopeModule)arg1, (str)arg2) -> None :
        Subscribe to one or several scope nodes. After subscription the scope
        module can be started with the 'execute' command. During the
        module run paths can not be subscribed or unsubscribed.
        arg1: Reference to the ScopeModule class.
        arg2: Path string of the node. Wildcards allowed.
               Alternatively also a list of path
               strings can be specified.

trigger(...)
    trigger( (ScopeModule)arg1) -> None :
        Not applicable to this module.

unsubscribe(...)
    unsubscribe( (ScopeModule)arg1, (str)arg2) -> None :
        Unsubscribe from one or several nodes. During the
        module run paths can not be subscribed or unsubscribed.
        arg1: Reference to the ScopeModule class.
        arg2: Path string of the node. Use wildcard to
               unsubscribe all. Alternatively also a list of path
               strings can be specified.

-----
Static methods inherited from Boost.Python.instance:

__new__(*args, **kwargs) from Boost.Python.class
    Create and return a new object. See help(type) for accurate signature.

-----
Data descriptors inherited from Boost.Python.instance:

__dict__

weakref

```

5.4.12. Help for the **SweeperModule** class

An instance of `SweeperModule` is initialized using the `sweep` method from `ziDAQServer`:

```
>>> help('zhinst.ziPython.ziDAQServer.sweep')

Help on built-in function sweep in zhinst.ziPython.ziDAQServer:

zhinst.ziPython.ziDAQServer.sweep = sweep(...)
  sweep( (ziDAQServer)arg1) -> SweeperModule :
    Create a sweeper class. This will start a thread for asynchronous
    sweeping.
      arg1: Reference to the ziDAQServer class.
      arg2: Timeout in [ms]. Recommended value is 500ms.
            DEPRECATED, ignored

  sweep( (ziDAQServer)arg1, (int)arg2) -> SweeperModule
```

Reference help for the SweeperModule class.

```
>>> help('zhinst.ziPython.SweeperModule')

Help on class SweeperModule in zhinst.ziPython:

zhinst.ziPython.SweeperModule = class SweeperModule(Boost.Python.instance)
|   Method resolution order:
|       SweeperModule
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       Raises an exception
|       This class cannot be instantiated from Python
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   clear(...)
|       clear( (SweeperModule)arg1) -> None :
|           End the sweeper thread.
|
|   execute(...)
|       execute( (SweeperModule)arg1) -> None :
|           Start the sweeper. Subscription or unsubscription is not
|           possible until the sweep is finished.
|
|   finish(...)
|       finish( (SweeperModule)arg1) -> None :
|           Stop sweeping. The sweeping may be restarted by calling
|           'execute' again.
|
|   finished(...)
|       finished( (SweeperModule)arg1) -> bool :
|           Check if the sweep has finished. Returns True if finished.
|
|   get(...)
|       get( (SweeperModule)arg1, (str)arg2, (bool)arg3) -> object :
|           Return a dict with all nodes from the specified sub-tree.
|           arg1: Reference to the SweeperModule class.
|           arg2: Path string of the node. Use wild card to
|               select all.
|           arg3[optional]: Specify which type of data structure to return.
|               Return data either as a flat dict (True) or as a nested
|               dict tree (False). Default = False.
|
|       get( (SweeperModule)arg1, (str)arg2) -> object
|
|   getDouble(...)
|       getDouble( (SweeperModule)arg1, (str)arg2) -> float :
```

```

        Return the floating point double value for the specified path.
        arg1: Reference to the SweeperModule class.
        arg2: Path string of the node.

getInt(...)
    getInt( (SweeperModule)arg1, (str)arg2) -> int :
        Return the integer value for the specified path.
        arg1: Reference to the SweeperModule class.
        arg2: Path string of the node.

getString(...)
    getString( (SweeperModule)arg1, (str)arg2) -> object :
        Return the string value for the specified path.
        arg1: Reference to the SweeperModule class.
        arg2: Path string of the node.

getStringUnicode(...)
    getStringUnicode( (SweeperModule)arg1, (str)arg2) -> object :
        Get a unicode string value from the specified node. The returned
        string is unicode encoded.
        Only relevant for Python versions older than V3.0.
        For Python versions 3.0 and later, getString can be used instead.
        arg1: Reference to the SweeperModule class.
        arg2: Path string of the node.

listNodes(...)
    listNodes( (SweeperModule)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the RecorderModule class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            ziPython.ziListEnum.none -> 0x00
                The default flag, returning a simple
                listing if the given node
            ziPython.ziListEnum.recursive -> 0x01
                Returns the nodes recursively
            ziPython.ziListEnum.absolute -> 0x02
                Returns absolute paths
            ziPython.ziListEnum.leafsonly -> 0x04
                Returns only nodes that are leafs,
                which means the they are at the
                outermost level of the tree.
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
        Or combinations of flags can be used.

progress(...)
    progress( (SweeperModule)arg1) -> object :
        Reports the progress of the measurement with a number between
        0 and 1.

read(...)
    read( (SweeperModule)arg1, (bool)arg2) -> object :
        Read sweep data. If the sweeping is still ongoing only a subset
        of sweep data is returned. If huge data sets are recorded call
        this method to keep memory usage reasonable.
        arg1[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

    read( (SweeperModule)arg1) -> object

save(...)
    save( (SweeperModule)arg1, (str)arg2) -> None :

```

```

        Save sweeper data to file.
        arg1: Reference to the SweeperModule class.
        arg2: File name string (without extension).

set(...)
set( (SweeperModule)arg1, (str)arg2, (float)arg3) -> None :
    Sweep Parameters
    Path name                Type    Description
    sweep/device              string  Device that should be used for
                                the parameter sweep, e.g. 'dev99'.
    sweep/start               double  Sweep start frequency [Hz]
    sweep/stop                double  Sweep stop frequency [Hz]
    sweep/gridnode            string  Path of the node that should be
                                used for sweeping. For frequency
                                sweep applications this will be e.g.
                                'oscs/0/freq'. The device name of
                                the path can be omitted and is given
                                by sweep/device.
    sweep/loopcount           int     Number of sweep loops (default 1)
    sweep/endsless            int     Sweep endless (default 0)
                                0 = endless off, use loopcount,
                                1 = endless on, ignore loopcount.
    sweep/samplecount         int     Number of samples per sweep.
    sweep/settling/time        double  Settling time before measurement is
                                performed.
    sweep/settling/tc          double  Shows the approximate settling precision
                                in time constant units as specified by
                                setting/inaccuracy (calculated upon
                                execute()). Setting this parameter
                                directly is now deprecated and may not
                                be supported in future versions.
                                5 ~ low precision
                                15 ~ medium precision
                                50 ~ high precision
    sweep/settling/inaccuracy  int     Demodulator filter settling inaccuracy
                                that defines the wait time between a
                                sweep parameter change and recording of
                                the next sweep point. The settling time
                                is calculated as the time required to
                                attain the specified remaining proportion
                                [1e-13, 0.1] of an incoming step
                                function.
                                Typical inaccuracy values:
                                10m ~ for highest sweep speed for large
                                    signals
                                100u ~ for precise amplitude measurements
                                100n ~ for precise noise measurements.
                                Depending on the order, the settling
                                inaccuracy will define the number of
                                filter time constants the sweeper has to
                                wait. The maximum between this value
                                and the settling time is taken as wait
                                time until the next sweep point is
                                recorded.
    sweep/xmapping            int     Sweep mode:
                                0 = linear,
                                1 = logarithmic.
    sweep/scan                int     Scan type:
                                0 = sequential,
                                1 = binary,
                                2 = bidirectional,
                                3 = reverse.
    sweep/bandwidth            double  Fixed bandwidth [Hz],
                                0 = Automatic calculation.
    sweep/bandwidthoverlap    int     Sets the bandwidth overlap mode,
                                (default 0):

```


		0 = disabled 1 = enabled
		If enabled the bandwidth of a sweep point may overlap with the frequency of neighboring sweep points. The effective bandwidth is only limited by the maximal bandwidth setting and omega suppression. As a result, the bandwidth is independent of the number of sweep points. For frequency response analysis bandwidth overlap should be enabled to achieve maximal sweep speed.
sweep/bandwidthcontrol	int	Sets the bandwidth control mode, (default 2): 0 = Manual (user sets bandwidth and order), 1 = Fixed (uses fixed bandwidth value), 2 = Auto (calculates best bandwidth value) Equivalent to the obsolete bandwidth = 0 setting.
sweep/order	int	Defines the filter roll off to use in Fixed bandwidth selection. Valid values are between 0 (6 dB/octave) and 8 (48 dB/octave). An order of 0 triggers a read-out of the order from the selected demodulator.
sweep/maxbandwidth	double	Maximal bandwidth used in auto bandwidth mode in [Hz]. The default is 1.25MHz.
sweep/omegasuppression	double	Damping in [dB] of omega and 2-omega components. Default is 40dB in favor of sweep speed. Use higher value for strong offset values or 3-omega measurement methods.
sweep/averaging/tc	double	Min averaging time [tc] 0 = no averaging (see also time!) 5 ~ low precision 15 ~ medium precision 50 ~ high precision
sweep/averaging/sample	int	Min samples to average 1 = no averaging (if averaging/tc = 0)
sweep/averaging/time	double	Min averaging time [s]
sweep/phaseunwrap	bool	Enable unwrapping of slowly changing phase evolutions around the +/-180 degree boundary.
sweep/sincfilter	bool	Enables the sinc filter if the sweep frequency is below 50 Hz. This will improve the sweep speed at low frequencies as omega components do not need to be suppressed by the normal low pass filter.
sweep/awgcontrol	bool	Enable AWG control for sweeper. If enabled the sweeper will automatically start the AWG and records the sweep sample based on the even index in hwtrigger.
sweep/save/directory	string	The base directory where files are saved.
sweep/save/filename	string	Defines the sub-directory where files are saved. The actual sub-directory has this name with an incrementing sequence count (per save) appended, e.g. sweep_000.
sweep/fileformat	string	The format of the file for saving sweeper measurements: 0 = Matlab, 1 = CSV, 2 = ZView (Impedance data only).
sweep/csvseparator	string	The character to use as CSV separator when saving files in this format.

sweep/csvlocale	string	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C" (default) = dot for the decimal point and no digit grouping, "" (empty string) = use the symbols set in the language and region settings of the computer.
sweep/save/save	bool	Initiate the saving of data to file. The saving is done in the background. When the save is finished, this parameter goes low.
sweep/save/saveonread	bool	Automatically save the data to file immediately before reading the data out of the module using the read() command. Set this parameter when you want to save data to file when running the module in continuous mode while doing intermediate reads.
sweep/historylength	bool	Sets an upper limit for the number of data captures stored in the module.
sweep/clearhistory	bool	Clear all captured data from the module.
set((SweeperModule)arg1, (str)arg2, (int)arg3) -> None		
set((SweeperModule)arg1, (str)arg2, (str)arg3) -> None		
set((SweeperModule)arg1, (object)arg2) -> None :		
arg1: Reference to the SweeperModule class.		
arg2: A list of path/value pairs.		
subscribe(...)		
subscribe((SweeperModule)arg1, (str)arg2) -> None :		
Subscribe to one or several nodes. After subscription the sweep process can be started with the 'execute' command. During the sweep process paths can not be subscribed or unsubscribed.		
arg1: Reference to the SweeperModule class.		
arg2: Path string of the node. Use wild card to select all. Alternatively also a list of path strings can be specified.		
trigger(...)		
trigger((SweeperModule)arg1) -> None :		
Execute a manual trigger.		
unsubscribe(...)		
unsubscribe((SweeperModule)arg1, (str)arg2) -> None :		
Unsubscribe from one or several nodes. During the sweep process paths can not be subscribed or unsubscribed.		
arg1: Reference to the SweeperModule class.		
arg2: Path string of the node. Use wild card to select all. Alternatively also a list of path strings can be specified.		

Static methods inherited from Boost.Python.instance:		
__new__(*args, **kwargs) from Boost.Python.class		
Create and return a new object. See help(type) for accurate signature.		

Data descriptors inherited from Boost.Python.instance:		
__dict__		
__weakref__		

5.4.13. Help for the **RecorderModule** class

An instance of RecorderModule is initialized using the record method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.record')
```

Help on built-in function record in zhinst.ziPython.ziDAQServer:

```
zhinst.ziPython.ziDAQServer.record = record(...)
record( (ziDAQServer)arg1) -> RecorderModule :
    Create a recording class. This will start a thread for asynchronous
    recording.
    arg1: Reference to the ziDAQServer class.
    arg2: Maximum recording time for single triggers in [s].
          DEPRECATED, set 'buffersize' param instead.
    arg3: Timeout in [ms]. Recommended value is 500ms.
          DEPRECATED, ignored.
    arg4[optional]: Record flags.
                    DEPRECATED, set 'flags' param instead.
                    FILL = 0x0001 : Fill holes.
                    ALIGN = 0x0002 : Align data that contains a
                                     timestamp.
                    THROW = 0x0004 : Throw EOFError exception if
                                     sample loss is detected.
                    DETECT = 0x0008: Detect data loss holes.

    record( (ziDAQServer)arg1, (float)arg2, (int)arg3 [, (int)arg4]) ->
    RecorderModule
```

Reference help for the RecorderModule class.

```
>>> help('zhinst.ziPython.RecorderModule')
```

Help on class RecorderModule in zhinst.ziPython:

```
zhinst.ziPython.RecorderModule = class RecorderModule(Boost.Python.instance)
| Method resolution order:
| RecorderModule
| Boost.Python.instance
| builtins.object
|
| Static methods defined here:
|
| __init__(...)
|     Raises an exception
|     This class cannot be instantiated from Python
|
| __reduce__ = <unnamed Boost.Python function>(...)
|
| clear(...)
|     clear( (RecorderModule)arg1) -> None :
|         End the recording thread.
|
| execute(...)
|     execute( (RecorderModule)arg1) -> None :
|         Start the recorder. After that command any trigger will start
|         the measurement. Subscription or unsubscription is not
|         possible until the recording is finished.
|
| finish(...)
|     finish( (RecorderModule)arg1) -> None :
|         Stop recording. The recording may be restarted by calling
|         'execute' again.
```

```

| finished(...)
|     finished( (RecorderModule)arg1) -> bool :
|         Check if the recording has finished. Returns True if finished.
|
| get(...)
|     get( (RecorderModule)arg1, (str)arg2, (bool)arg3) -> object :
|         Return a dict with all nodes from the specified sub-tree.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node. Use wild card to
|             select all.
|         arg3[optional]: Specify which type of data structure to return.
|             Return data either as a flat dict (True) or as a nested
|             dict tree (False). Default = False.
|
|     get( (RecorderModule)arg1, (str)arg2) -> object
|
| getDouble(...)
|     getDouble( (RecorderModule)arg1, (str)arg2) -> float :
|         Return the floating point double value for the specified path.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node.
|
| getInt(...)
|     getInt( (RecorderModule)arg1, (str)arg2) -> int :
|         Return the integer value for the specified path.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node.
|
| getString(...)
|     getString( (RecorderModule)arg1, (str)arg2) -> object :
|         Return the string value for the specified path.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node.
|
| getStringUnicode(...)
|     getStringUnicode( (RecorderModule)arg1, (str)arg2) -> object :
|         Return the unicode encoded string value for the specified path.
|         Only relevant for Python versions older than V3.0.
|         For Python versions 3.0 and later, getString can be used instead.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node.
|
| listNodes(...)
|     listNodes( (RecorderModule)arg1, (str)arg2, (int)arg3) -> list :
|         This function returns a list of node names found at the specified path.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path for which the nodes should be listed. The path may
|             contain wildcards so that the returned nodes do not
|             necessarily have to have the same parents.
|         arg3: Enum that specifies how the selected nodes are listed.
|             ziPython.ziListEnum.none -> 0x00
|                 The default flag, returning a simple
|                 listing if the given node
|             ziPython.ziListEnum.recursive -> 0x01
|                 Returns the nodes recursively
|             ziPython.ziListEnum.absolute -> 0x02
|                 Returns absolute paths
|             ziPython.ziListEnum.leafsonly -> 0x04
|                 Returns only nodes that are leafs,
|                 which means the they are at the
|                 outermost level of the tree.
|             ziPython.ziListEnum.settingsonly -> 0x08
|                 Returns only nodes which are marked
|                 as setting
|         Or combinations of flags can be used.
|
| progress(...)

```

```

progress( (RecorderModule)arg1) -> object :
    Reports the progress of the measurement with a number between
    0 and 1.

read(...)
    read( (RecorderModule)arg1, (bool)arg2) -> object :
        Read recorded data. If the recording is still ongoing only a subset
        of recorded data is returned. If many triggers or huge data sets
        are recorded call this method to keep memory usage reasonable.
        arg1[optional]: Specify which type of data structure to return.
        Return data either as a flat dict (True) or as a nested
        dict tree (False). Default = False.

    read( (RecorderModule)arg1) -> object

save(...)
    save( (RecorderModule)arg1, (str)arg2) -> None :
        Save trigger data to file.
        arg1: Reference to the RecorderModule class.
        arg2: File name string (without extension).

set(...)
    set( (RecorderModule)arg1, (str)arg2, (float)arg3) -> None :
        Software Trigger Module Parameters
        Path name      Type      Description
        trigger/buffersize  double  Overwrite the buffersize [s] of the
                                trigger object (set when it was
                                instantiated). Recommended buffer size
                                is 2*trigger/0/duration.

        trigger/flags      int      Record flags.
                                FILL = 0x0001 : Fill holes.
                                ALIGN = 0x0002 : Align data that contains
                                                a timestamp.
                                THROW = 0x0004 : Throw EOFError exception
                                                if sample loss is
                                                detected.
                                DETECT = 0x0008: Detect data loss holes.

        trigger/device      string   The device ID to execute the software
                                trigger, e.g. dev123

        trigger/endless     bool     Enable endless triggering:
                                1 = enable,
                                0 = disable.

        trigger/forcetrigger bool     Force a trigger.

        trigger/awgcontrol  bool     Enable interaction with AWG. If enabled
                                the hwtrigger index counter will be used
                                to control the grid row for recording.

        trigger/0/triggernode string Path and signal of the node that should
                                be used for triggering, separated by a
                                dot (.), e.g. /devN/demods/0/sample.x
                                Overrides values from trigger/0/path and
                                trigger/0/source.

        trigger/0/path      string   The path to the demod sample to trigger
                                on, e.g. demods/3/sample, see also
                                trigger/0/source.
                                DEPRECATED - use trigger/0/triggernode
                                instead.

        trigger/0/source    int      Signal that is used to trigger on.
                                0 = x [X_SOURCE]
                                1 = y [Y_SOURCE]
                                2 = r [R_SOURCE]
                                3 = angle [ANGLE_SOURCE]
                                4 = frequency [FREQUENCY_SOURCE]
                                5 = phase [PHASE_SOURCE]
                                6 = auxiliary input 0 / parameter 0
                                    [AUXIN0_SOURCE / PARAM0_SOURCE]
                                7 = auxiliary input 1 / parameter 1
                                    [AUXIN1_SOURCE / PARAM1_SOURCE]

```

		DEPRECATED - use trigger/0/triggernode instead.
trigger/0/count	int	Number of trigger edges to record.
trigger/0/type	int	Trigger type used. Some parameters are only valid for special trigger nodes and/or types. 0 = trigger off 1 = analog edge trigger on source 2 = digital trigger mode on DIO 3 = analog pulse trigger on source 4 = analog tracking trigger on source 5 = change trigger 6 = hardware trigger on trigger line source. 7 = tracking edge trigger on source 8 = event count trigger on counter source.
trigger/0/edge	int	Trigger edge 1 = rising edge 2 = falling edge 3 = both
trigger/0/findlevel	bool	Automatically find the value of trigger/0/level based on the current signal value.
trigger/0/bits	int	Digital trigger condition.
trigger/0/bitmask	int	Bit masking for bits used for triggering. Used for digital trigger.
trigger/0/delay	double	Trigger frame position [s] (left side) relative to trigger edge. delay = 0 -> trigger edge at left border. delay < 0 -> trigger edge inside trigger frame (pretrigger). delay > 0 -> trigger edge before trigger frame (posttrigger).
trigger/0/duration	double	Recording frame length [s].
trigger/0/level	double	Trigger level voltage [V].
trigger/0/hysteresis	double	Trigger hysteresis [V].
trigger/0/retrigger	int	Record more than one trigger in a trigger frame. If a trigger event is currently being recorded and a new trigger event is detected within the duration of the current trigger event, extend the size of the trigger frame to include the duration of the new trigger event.
trigger/triggered	bool	Has the software trigger triggered? 1=Yes, 0=No (read-only).
trigger/0/bandwidth	double	Filter bandwidth [Hz] for pulse and tracking triggers.
trigger/0/holdoff/count	int	Number of skipped triggers until the next trigger is recorded again.
trigger/0/holdoff/time	double	Hold off time [s] before the next trigger is recorded again. A hold off time smaller than the duration will produce overlapped trigger frames.
trigger/0/hwtrigsources	int	Only available for devices that support hardware triggering. Specify the channel to trigger on. DEPRECATED - use trigger/0/triggernode instead.
trigger/0/pulse/min	double	Minimal pulse width [s] for the pulse trigger.
trigger/0/pulse/max	double	Maximal pulse width [s] for the pulse trigger.
trigger/0/eventcount/mode	int	Specifies the mode used for event count processing. 0 - Trigger on every event count sample.

		1 - Trigger if event count value incremented.
trigger/0/grid/mode	int	Enable grid mode. In grid mode a matrix instead of a vector is returned. Each trigger becomes a row in the matrix and each trigger's data is interpolated onto a new grid defined by the number of columns: 0: Disable 1: Enable with nearest neighbour interpolation. 2: Enable with linear interpolation.
trigger/0/grid/operation	int	If running in endless mode, either replace or average the data in the grid's matrix.
trigger/0/grid/cols	int	Specify the number of columns in the grid's matrix. The data from each row is interpolated onto a grid with the specified number of columns.
trigger/0/grid/rows	int	Specify the number of rows in the grid's matrix. Each row is the data recorded from one trigger interpolated onto the columns.
trigger/0/grid/repetitions	int	Number of statistical operations performed per grid.
trigger/0/grid/direction	int	The direction to organize data in the grid's matrix: 0: Forward. The data in each row is ordered chronologically, e.g., the first data point in each row corresponds to the first timestamp in the trigger data. 1: Reverse. The data in each row is ordered reverse chronologically, e.g., the first data point in each row corresponds to the last timestamp in the trigger data. 2: Bidirectional. The ordering of the data alternates between Forward and Backward ordering from row-to-row. The first row is Forward ordered.
trigger/save/directory	string	The base directory where files are saved.
trigger/save/filename	string	Defines the sub-directory where files are saved. The actual sub-directory has this name with an incrementing sequence count (per save) appended , e.g. swtrigger_000.
trigger/save/fileformat	string	The format of the file for saving data: 0 = Matlab, 1 = CSV, 2 = ZView (Impedance data only).
trigger/save/csvseparator	string	The character to use as CSV separator when saving files in this format.
trigger/save/csvlocale	string	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C" (default) = dot for the decimal point and no digit grouping, "" (empty string) = use the symbols set in the language and region settings of the computer.
trigger/save/save	bool	Initiate the saving of data to file. The saving is done in the background. When the save is finished, this parameter goes low.
trigger/save/saveonread	bool	Automatically save the data to file immediately before reading the data

```

|                                     out of the module using the read()
|                                     command. Set this parameter when you
|                                     want to save data to file when
|                                     running the module in continous mode
|                                     while doing intermediate reads.
|
|         trigger/historylength      int      Sets an upper limit for the number of
|
|         trigger/clearhistory       bool     Clear all captured data from the module.
|
|
|     set( (RecorderModule)arg1, (str)arg2, (int)arg3) -> None
|
|     set( (RecorderModule)arg1, (str)arg2, (str)arg3) -> None
|
|     set( (RecorderModule)arg1, (object)arg2) -> None :
|         arg1: Reference to the RecorderModule class.
|         arg2: A list of path/value pairs.
|
| subscribe(...)
|     subscribe( (RecorderModule)arg1, (str)arg2) -> None :
|         Subscribe to one or several nodes. After subscription the recording
|         process can be started with the 'execute' command. During the
|         recording process paths can not be subscribed or unsubscribed.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node. Use wild card to
|               select all. Alternatively also a list of path
|               strings can be specified.
|
| trigger(...)
|     trigger( (RecorderModule)arg1) -> None :
|         Execute a manual trigger.
|
| unsubscribe(...)
|     unsubscribe( (RecorderModule)arg1, (str)arg2) -> None :
|         Unsubscribe from one or several nodes. During the
|         recording process paths can not be subscribed or unsubscribed.
|         arg1: Reference to the RecorderModule class.
|         arg2: Path string of the node. Use wild card to
|               select all. Alternatively also a list of path
|               strings can be specified.
|
| -----
| Static methods inherited from Boost.Python.instance:
|
| __new__(*args, **kwargs) from Boost.Python.class
|     Create and return a new object. See help(type) for accurate signature.
|
| -----
| Data descriptors inherited from Boost.Python.instance:
|
| __dict__
|
| __weakref__

```

5.4.14. Help for the **ZoomFFTModule** class

An instance of ZoomFFTModule is initialized using the zoomFFT method from ziDAQServer:

```
>>> help('zhinst.ziPython.ziDAQServer.zoomFFT')
```

Help on built-in function zoomFFT in zhinst.ziPython.ziDAQServer:

```

zhinst.ziPython.ziDAQServer.zoomFFT = zoomFFT(...)
zoomFFT( (ziDAQServer)arg1) -> ZoomFFTModule :
    Create a ZoomFFTModule class. This will start a thread for running an
    asynchronous ZoomFFTModule.

```


arg1: Reference to the ziDAQServer class.
arg2: Timeout in [ms]. Recommended value is 500ms.
DEPRECATED, ignored

zoomFFT((ziDAQServer)arg1, (int)arg2) -> ZoomFFTModule

Reference help for the ZoomFFTModule class.

```
>>> help('zhinst.ziPython.ZoomFFTModule')
```

Help on class ZoomFFTModule in zhinst.ziPython:

```
zhinst.ziPython.ZoomFFTModule = class ZoomFFTModule(Boost.Python.instance)
|   Method resolution order:
|       ZoomFFTModule
|       Boost.Python.instance
|       builtins.object
|
|   Static methods defined here:
|
|   __init__(...)
|       Raises an exception
|       This class cannot be instantiated from Python
|
|   __reduce__ = <unnamed Boost.Python function>(...)
|
|   clear(...)
|       clear( (ZoomFFTModule)arg1) -> None :
|           End the zoom FFT thread.
|
|   execute(...)
|       execute( (ZoomFFTModule)arg1) -> None :
|           Start the zoom FFT. Subscription or unsubscription is not
|           possible until the zoom FFT is finished.
|
|   finish(...)
|       finish( (ZoomFFTModule)arg1) -> None :
|           Stop the zoom FFT. The zoom FFT may be restarted by calling
|           'execute' again.
|
|   finished(...)
|       finished( (ZoomFFTModule)arg1) -> bool :
|           Check if the zoom FFT has finished. Returns True if finished.
|
|   get(...)
|       get( (ZoomFFTModule)arg1, (str)arg2, (bool)arg3) -> object :
|           Return a dict with all nodes from the specified sub-tree.
|           arg1: Reference to the ZoomFFTModule class.
|           arg2: Path string of the node. Use wild card to
|               select all.
|           arg3[optional]: Specify which type of data structure to return.
|               Return data either as a flat dict (True) or as a nested
|               dict tree (False). Default = False.
|
|       get( (ZoomFFTModule)arg1, (str)arg2) -> object
|
|   getDouble(...)
|       getDouble( (ZoomFFTModule)arg1, (str)arg2) -> float :
|           Return the floating point double value for the specified path.
|           arg1: Reference to the ZoomFFTModule class.
|           arg2: Path string of the node.
|
|   getInt(...)
|       getInt( (ZoomFFTModule)arg1, (str)arg2) -> int :
|           Return the integer value for the specified path.
|           arg1: Reference to the ZoomFFTModule class.
```

```

        arg2: Path string of the node.

getString(...)
    getString( (ZoomFFTModule)arg1, (str)arg2) -> object :
        Return the string value for the specified path.
        arg1: Reference to the ZoomFFTModule class.
        arg2: Path string of the node.

getStringUnicode(...)
    getStringUnicode( (ZoomFFTModule)arg1, (str)arg2) -> object :
        Get a unicode string value from the specified node.
        The returned string is unicode encoded.
        Only relevant for Python versions older than V3.0.
        For Python versions 3.0 and later, getString can be used instead.
        arg1: Reference to the ZoomFFTModule class.
        arg2: Path string of the node.

listNodes(...)
    listNodes( (ZoomFFTModule)arg1, (str)arg2, (int)arg3) -> list :
        This function returns a list of node names found at the specified path.
        arg1: Reference to the ZoomFFTModule class.
        arg2: Path for which the nodes should be listed. The path may
            contain wildcards so that the returned nodes do not
            necessarily have to have the same parents.
        arg3: Enum that specifies how the selected nodes are listed.
            ziPython.ziListEnum.none -> 0x00
                The default flag, returning a simple
                listing if the given node
            ziPython.ziListEnum.recursive -> 0x01
                Returns the nodes recursively
            ziPython.ziListEnum.absolute -> 0x02
                Returns absolute paths
            ziPython.ziListEnum.leafsonly -> 0x04
                Returns only nodes that are leafs,
                which means the they are at the
                outermost level of the tree.
            ziPython.ziListEnum.settingsonly -> 0x08
                Returns only nodes which are marked
                as setting
            Or combinations of flags can be used.

progress(...)
    progress( (ZoomFFTModule)arg1) -> object :
        Reports the progress of the measurement with a number between
        0 and 1.

read(...)
    read( (ZoomFFTModule)arg1, (bool)arg2) -> object :
        Read zoom FFT data. If the zoom FFT is still ongoing only a subset
        of zoom FFT data is returned.
        arg1[optional]: Specify which type of data structure to return.
            Return data either as a flat dict (True) or as a nested
            dict tree (False). Default = False.

    read( (ZoomFFTModule)arg1) -> object

save(...)
    save( (ZoomFFTModule)arg1, (str)arg2) -> None :
        Save zoom FFT data to file.
        arg1: Reference to the ZoomFFTModule class.
        arg2: File name string (without extension).

set(...)
    set( (ZoomFFTModule)arg1, (str)arg2, (float)arg3) -> None :
        Zoom FFT Parameters
        Path name      Type      Description
        zoomFFT/device  string    Device that should be used for

```

		the FFT, e.g. 'dev99'.
zoomFFT/bit	int	Number of FFT points 2^{bit}
zoomFFT/mode	int	Zoom FFT mode
		0 = Perform FFT on X+iY
		1 = Perform FFT on R
		2 = Perform FFT on Phase
		3 = Perform FFT on Frequency
		4 = Perform FFT on Phase derivative
zoomFFT/loopcount	int	Number of zoomFFT loops (default 1)
zoomFFT/endless	int	Run the FFT continuously (default 0)
		0 = endless off, use loopcount
		1 = endless on, ignore loopcount
zoomFFT/overlap	double	Overlap of the demod data used for the FFT overlap 0 = none, [0..1]
zoomFFT/settling/time	double	Settling time before measurement is performed.
zoomFFT/settling/tc	double	Settling time in time constant units before the FFT recording is started.
		5 ~ low precision
		15 ~ medium precision
		50 ~ high precision
zoomFFT/window	int	FFT window (default 1 = Hann)
		0 = Rectangular
		1 = Hann
		2 = Hamming
		3 = Blackman Harris 4 term
		16 = Exponential (ring-down)
		17 = Cosine (ring-down)
		18 = Cosine squared (ring-down)
zoomFFT/absolute	bool	Shifts the frequencies so that the center frequency becomes the demodulation frequency rather than 0 Hz.
zoomFFT/save/directory	string	The base directory where files are saved.
zoomFFT/save/filename	string	Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count appended, e.g. zoomFFT_000.
zoomFFT/fileformat	string	The format of the file for saving sweeper measurements:
		0 = Matlab,
		1 = CSV,
		2 = ZView (Impedance data only).
zoomFFT/csvseparator	string	The character to use as CSV separator when saving files in this format.
zoomFFT/csvlocale	string	The locale to use for the decimal point character and digit grouping character for numerical values in CSV files:
		"C" (default) = dot for the decimal point and no digit grouping,
		"" (empty string) = use the symbols set in the language and region settings of the computer.
zoomFFT/save/save	bool	Initiate the saving of data to file. The saving is done in the background. When the save is finished, this parameter goes low.
zoomFFT/save/saveonread	bool	Automatically save the data to file immediately before reading the data out of the module using the read() command. Set this parameter when you want to save data to file when running the module in continuous mode while doing intermediate reads.
set((ZoomFFTModule)arg1, (str)arg2, (int)arg3) -> None		

```
set( (ZoomFFTModule)arg1, (str)arg2, (str)arg3) -> None

set( (ZoomFFTModule)arg1, (object)arg2) -> None :
    arg1: Reference to the ZoomFFTModule class.
    arg2: A list of path/value pairs.

subscribe(...)
    subscribe( (ZoomFFTModule)arg1, (str)arg2) -> None :
        Subscribe to one or several nodes. After subscription the zoom FFT
        process can be started with the 'execute' command. During the
        zoom FFT process paths can not be subscribed or unsubscribed.
        arg1: Reference to the ZoomFFTModule class.
        arg2: Path string of the node. Use wild card to
              select all. Alternatively also a list of path
              strings can be specified.

trigger(...)
    trigger( (ZoomFFTModule)arg1) -> None :
        Execute a manual trigger.

unsubscribe(...)
    unsubscribe( (ZoomFFTModule)arg1, (str)arg2) -> None :
        Unsubscribe from one or several nodes. During the
        zoom FFT process paths can not be subscribed or unsubscribed.
        arg1: Reference to the ZoomFFTModule class.
        arg2: Path string of the node. Use wild card to
              select all. Alternatively also a list of path
              strings can be specified.

-----
Static methods inherited from Boost.Python.instance:

__new__(*args, **kwargs) from Boost.Python.class
    Create and return a new object. See help(type) for accurate signature.

-----
Data descriptors inherited from Boost.Python.instance:

__dict__
__weakref__
```

Chapter 6. LabVIEW Programming

Interfacing with your Zurich Instruments device via National Instruments' [LabVIEW®](#) is an efficient choice in terms of development time and run-time performance. LabVIEW is a graphical programming language designed to interface with laboratory equipment via so-called VIs ("virtual instruments"), whose key strength is the ease of displaying dynamic signals obtained from your instrument.

This chapter aims to help you get started using the Zurich Instruments LabOne LabVIEW API to control your instrument, please refer to:

- [Section 6.1](#) for help [Installing the LabOne LabVIEW API](#).
- [Section 6.2](#) for help [Getting Started with the LabOne LabVIEW API](#) and [running the examples](#).
- [Section 6.3](#) for some [LabVIEW Programming Tips and Tricks](#).

Note

This section and the provided examples are not intended to be a general LabVIEW tutorial. See, for example, the National Instruments [website](#) for help to get started programming with LabVIEW.

6.1. Installing the LabOne LabVIEW API

6.1.1. Requirements

One of the following platforms and LabVIEW versions is required to use the LabOne LabVIEW API:

1. 32 or 64-bit Windows with LabVIEW 2009 or newer.
2. 32 or 64-bit Linux with LabVIEW 2009 or newer.
3. 32 or 64-bit Mac OS X and LabVIEW 2010 or newer.

The LabOne LabVIEW API is included in a standard LabOne installation and is also available as a separate package (see below, [Separate LabVIEW Package](#)). In order to make the LabOne LabVIEW API available for use within LabVIEW, a directory needs to be copied to a specific directory of your LabVIEW installation. Both the main LabOne installer and the separate LabOne LabVIEW API package are available from Zurich Instruments' [download page](#).

Separate LabVIEW Package

The separate LabVIEW API package should be used if you would like to either:

1. Use the LabVIEW API on Mac OS X (the main LabOne installer is not available for Mac OS X).
2. Use the LabVIEW API to work with an instrument remotely (i.e., on a separate PC from where the Data Server is running) and you do not require a full LabOne installation. This is the case, for example, with MF Instruments.

Mac OS X

Since the Data Server is not available for use on Mac OS X, using the LabVIEW API on OS X requires a Data Server running remotely on either an instrument or a separate Windows or Linux PC. If the Data Server is running on a PC, please ensure that it's configured to accept connections from other PCs in the network as described in [Section 1.4.1](#).

6.1.2. Windows Installation

1. Locate the `instr.lib` directory in your LabVIEW installation and delete any previous Zurich Instruments API directories. The `instr.lib` directory is typically located at:

```
C:\Program Files\National Instruments\LabVIEW 201x\instr.lib\
```

Previous Zurich Instruments installations will be directories located in the `instr.lib` directory that are named either:

- Zurich Instruments HF2, or
- Zurich Instruments LabOne.

These folders may simply be deleted (administrator rights required).

2. On Windows, either navigate to the `API\LabVIEW` subdirectory of your LabOne installation or, in the case of the separate installer (see [Separate LabVIEW Package](#)), the directory of the unzipped LabOne LabVIEW package, and copy the subdirectory

```
Zurich Instruments LabOne
```

to the `instr.lib` directory in your LabVIEW installation as located in Step 1. Note, you will need administrator rights to copy to this directory.

In the case of copying from a LabOne installation, this folder is typically located at:

```
C:\Program Files\Zurich Instruments\LabOne\API\LabVIEW\
```

3. Restart LabVIEW and verify your installation as described in [Section 6.1.4](#).

6.1.3. Linux and Mac Installation

1. Locate the `instr.lib` directory in your LabVIEW installation and remove any previous Zurich Instruments API installations. The `instr.lib` directory is typically located on Linux at:

```
/usr/local/natinst/LabVIEW-201x/instr.lib/
```

and on Mac OS X at:

```
/Applications/National Instruments/LabVIEW 201x/instr.lib/
```

Previous Zurich Instruments installations will be folders located in the `instr.lib` directory that are named either:

- Zurich Instruments HF2, or
- Zurich Instruments LabOne.

These folders may simply be deleted (administrator rights required).

2. Navigate to the path where you unpacked LabOne or the [Separate LabVIEW Package](#) and copy the subdirectory

```
Zurich Instruments LabOne/
```

to the `instr.lib` directory in your LabVIEW installation as located in Step 1. Note, you will need administrator rights to copy to this directory.

Note, when copying from the main LabOne tarball (Linux only), the `Zurich Instruments LabOne/` directory is located in

```
[PATH]/LabOneLinux64/API/LabVIEW/
```

3. Restart LabVIEW and verify your installation as described in [Section 6.1.4](#).

6.1.4. Verifying your Installation

If the LabOne LabVIEW API palette can be accessed from within LabVIEW, the LabOne LabVIEW API is correctly installed. See [Section 6.2.1](#) for help finding the palette.

6.2. Getting Started with the LabOne LabVIEW API

6.2.1. Locating the LabOne LabVIEW VI Palette

In order to locate the LabOne LabVIEW VIs start LabVIEW and create a new VI. In the VI's "Block Diagram" (CTRL-e) you can to access the LabOne LabVIEW API palette with a mouse right-click and browsing the tree under "Instrument I/O" → "Instr. Drivers", see [Figure 6.1](#).

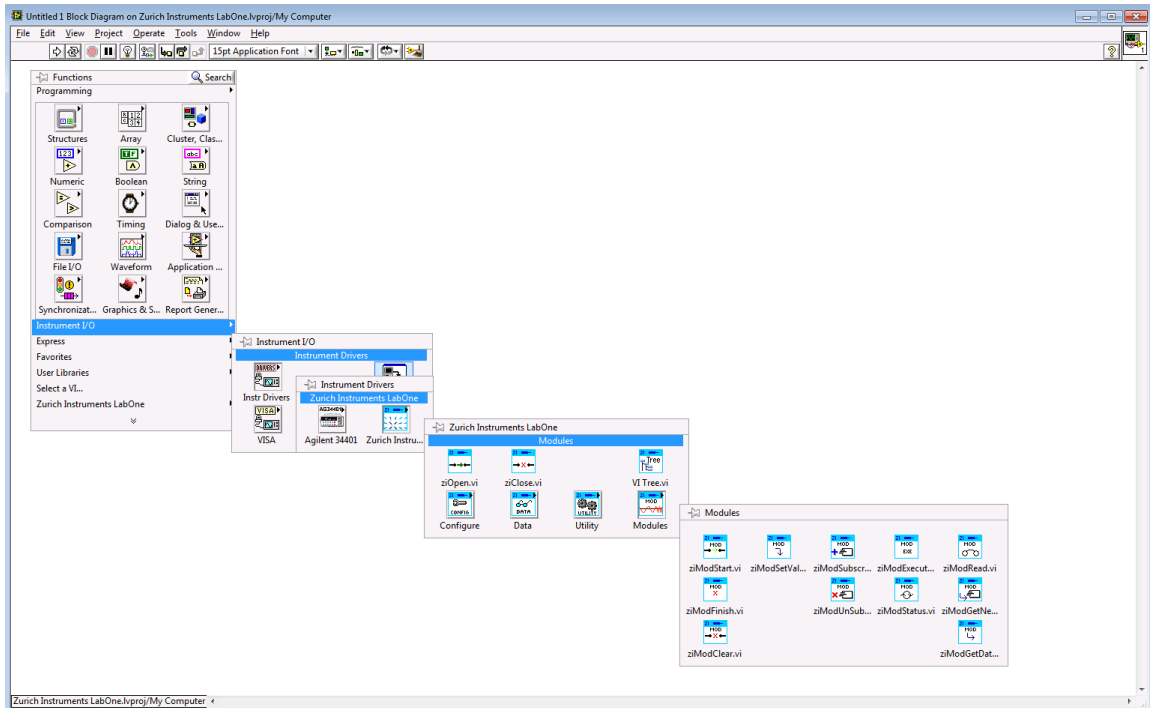


Figure 6.1. Locating the LabOne LabVIEW Palette

6.2.2. LabOne LabVIEW Programming Concepts

As described in [Section 1.2](#) a LabVIEW program communicates to a Zurich Instrument device via a software program running on the PC called the data server. In general, the outline of the instruction flow for a LabVIEW virtual instrument is as following:

1. Initialization: Open a connection from the API to the data server program.
2. Configuration: Perform the instrument's settings. For example, using the virtual instrument `ziSetValueDouble.vi`.
3. Data: Read data from the instrument.
4. Utility: Perform data analysis on the read data, potentially repeating Step 2 and/or Step 3.
5. Close: Terminate the API's connection to the data server program.

The `VI Tree.vi` included the LabOne LabVIEW API demonstrates this flow and lists common VIs used for working with a Zurich Instruments device, see [Figure 6.2](#). The `VI Tree.vi` can be found either via the LabOne VI palette, see [Section 6.2.1](#), or by opening the file in the `Public` folder of your LabOne LabVIEW installation, typically located at:

```
C:\Program Files\National Instruments\LabVIEW 2012\instr.lib\Zurich
Instruments LabOne\Public\VI Tree.vi
```

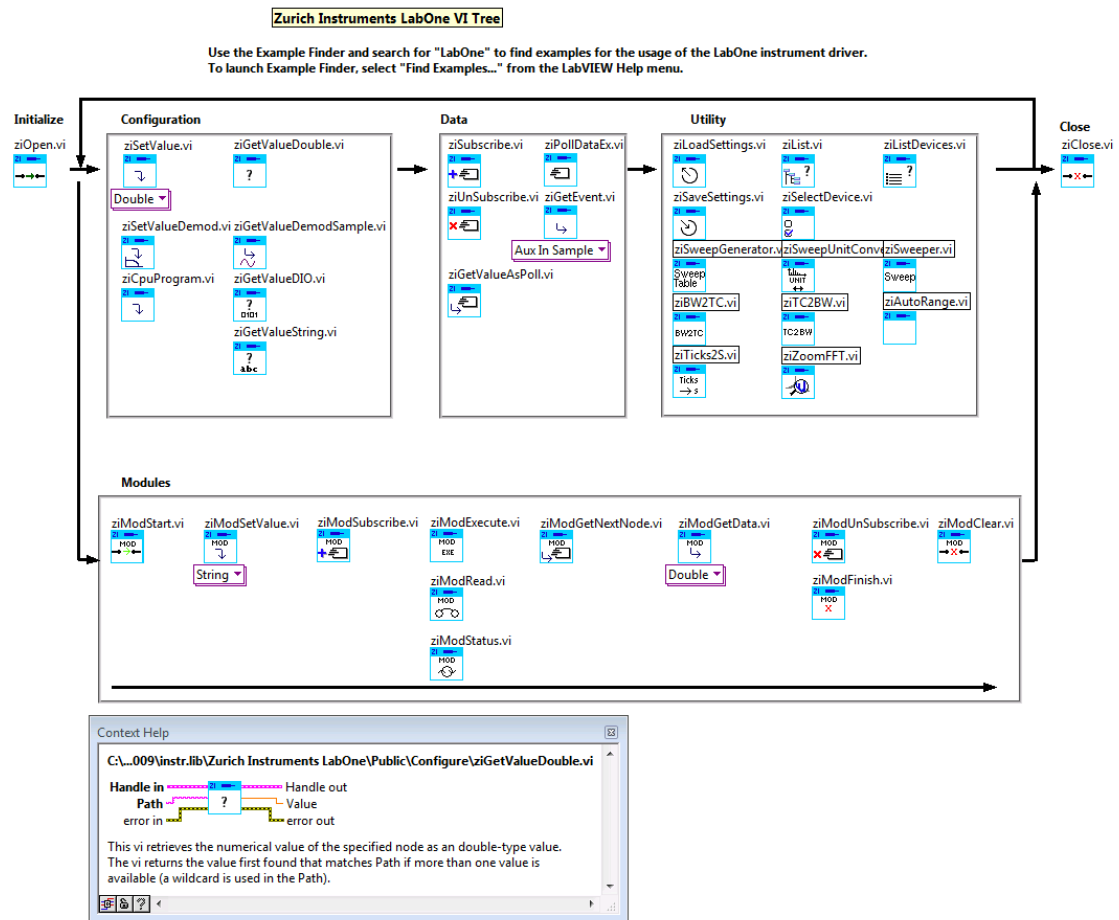



Figure 6.2. An overview of the LabOne LabVIEW VIs is given in VI Tree.vi. Press CTRL-h after selecting one of the VIs to obtain help.

6.2.3. Using ziCore Modules in the LabOne LabVIEW API

LabOne ziCore Modules (e.g. Sweeper) enable high-level measurement tools to use with your Zurich instrument device in LabVIEW. The outline of the instruction flow for a LabVIEW Module is as following:

1. Initialization: Create a ziModHandle from a ziHandle ziModStart.vi.
2. Configuration: Perform the module's settings. For example, using the virtual instrument ziModSetValue.vi.
3. Subscribe: Define the recorded data node ziModSubscribe.vi.
4. Execute: Start the operation of the module ziModExecute.vi.
5. Data: Read data from the module. For example, using the ziModGetNextNode.vi and ziModGetData.vi.
6. Utility: Perform data analysis on the read data, potentially repeating Step 2, Step 3 and/or Step 4.
7. Clear: Terminate the API's connection to the module ziModClear.vi.

6.2.4. Finding help for the LabOne VIs from within LabVIEW

As is customary for LabVIEW, built-in help for LabOne's VIs can be obtained by selecting the VI with the mouse in a block diagram and pressing CTRL-h to view the VI's context help. See [Figure 6.2](#) for an example.

6.2.5. Finding the LabOne LabVIEW API Examples

Many examples come bundled with the LabOne LabVIEW API which demonstrate the most important concepts of working with Zurich Instrument devices. The easiest way to browse the list of available examples is via the NI Example Finder: In LabVIEW select "Find Examples..." from the "Help" menu-bar and search for "LabOne", see [Figure 6.3](#).

The examples are located in the directory `instr.lib/Zurich Instruments LabOne/Examples` found in LabVIEW installation directory. In order to modify an example for your needs, please copy it to your local workspace.

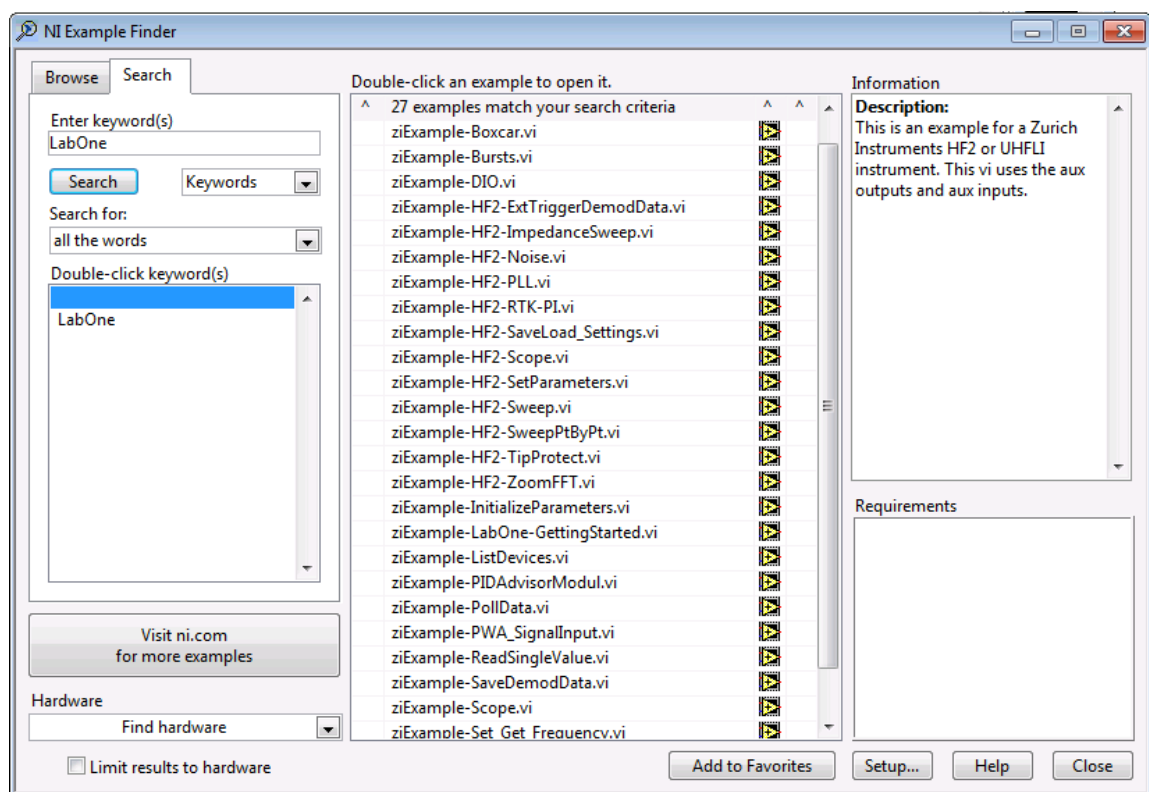


Figure 6.3. Search for "LabOne" in NI's Example Finder to find examples to run with your instrument.

6.2.6. Running the LabOne Example VIs

This section describes how to run a LabOne LabVIEW example on your instrument.

Note

Please ensure that the example you would like to run is supported by your instrument class and its options set. For example, examples for HF2 Instruments can be found in the Example Finder

(see [Section 6.2.5](#)) by searching for "HF2", examples for the UHFLI by searching for "UHFLI" and examples for the MFLI by searching for "MFLI".

Device Connection

After opening one of the LabOne LabVIEW examples, please ensure that the example is configured to run on the desired instrument type. `ziOpen.vi` establishes a connection to a Data Server. The address is of the format `{<host>}{:<port>}:{<Device ID>}`. Usually it is sufficient to provide the Device ID only highlighted in [Figure 6.4](#). The Device ID corresponds to the serial number (S/N) found on the instrument rear panel. The host and port are then determined by network discovery. Should the discovery not work, prepend `<host>:<port>::` to the Device ID. Examples are "myhf2.company.com:8004::dev466" or "myhf2.company.com:8004". In the latter case the first found instrument on the data server listening on "myhf2.company.com:8004" will be selected.

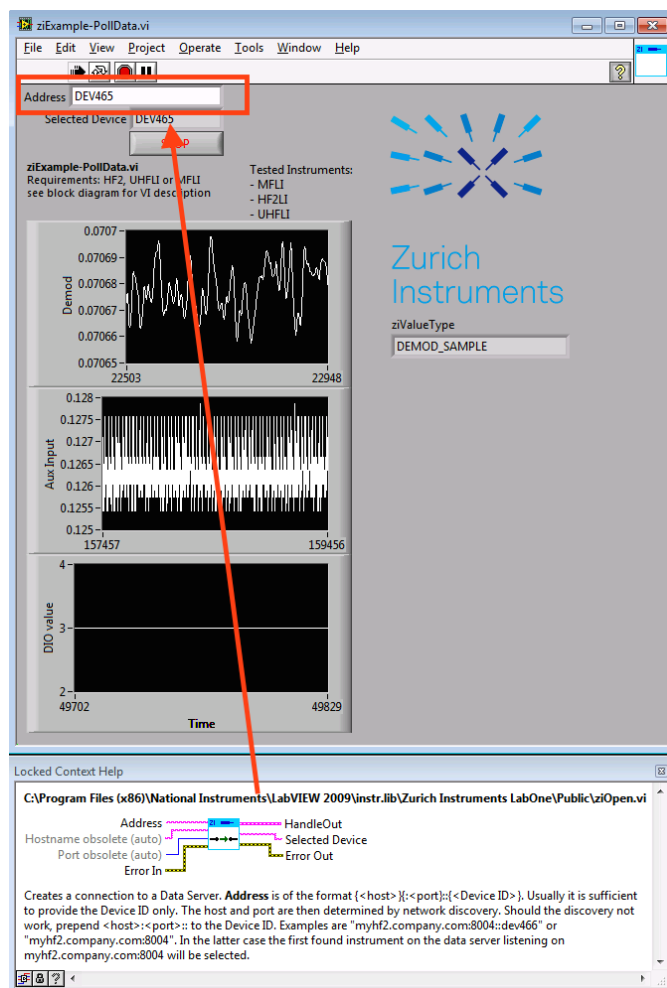


Figure 6.4. LabOne LabVIEW Example Poll Data: Device selection.

Running the VI and Block Diagram

The example can be ran as any LabVIEW program; by clicking the "Run" icon in the icon bar. Be sure to check the example's code and explanation by pressing CTRL-e to view the example's block diagram, see [Figure 6.5](#).

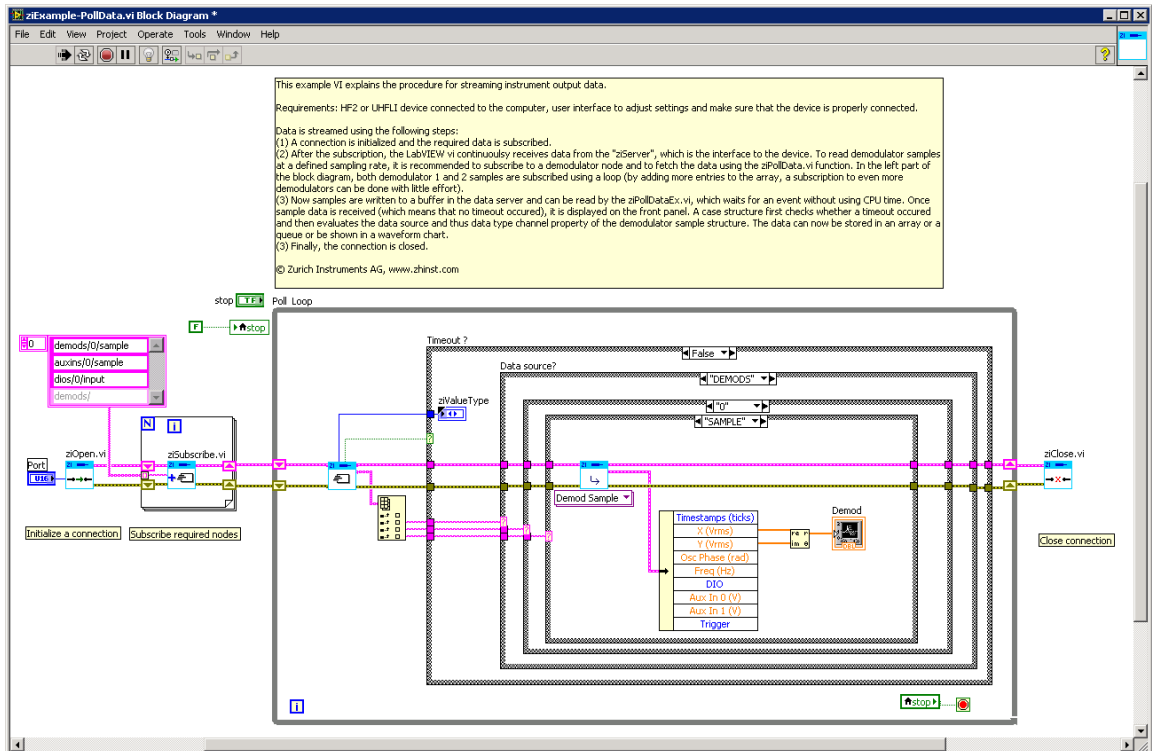


Figure 6.5. LabOne LabVIEW Example Poll Data: Block Diagram.

6.3. LabVIEW Programming Tips and Tricks

Use the User Interface's command log or Server's text interface while programming with LabVIEW

As with all other interfaces, LabVIEW uses the "path" and "nodes" concept to address settings on an instrument, see [Section 1.2](#). In order to learn about or verify the nodes available it can be very helpful to view the command log in the User Interface (see the bar in the bottom of the screen) to see which node has been configured during a previous setting change. The text interface (HF2 Series) provides a convenient way to explore the node hierarchy.

Always close ziHandles and ziModHandles or LabVIEW runs out of memory

If you use the "Abort Execution" button of LabVIEW, your LabVIEW program will not close any existing connections to the ziServer. Any open connection inside of LabVIEW will persist and continue to consume about 12 MB of RAM so that with time you will run out of memory. Completely exit LabVIEW in order to release the memory again.

Use shift registers

The structure of efficient LabVIEW code is distinguished by signals being "piped through" by use of shift registers in loops and by the absence of object replication. Using shift registers in LabVIEW avoids copying of data and, more important, running the garbage collector frequently.

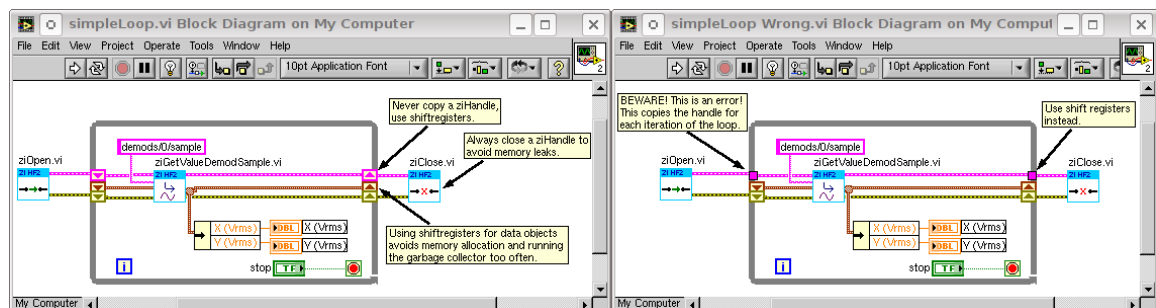


Figure 6.6. Examples of simple LabVIEW programs for the Zurich Instruments HF2 Series. Left: A well implemented loop, Right: An example for-loop gone wrong.

Chapter 7. .NET Programming

This chapter helps you get started using Zurich Instruments LabOne's .NET API to control your instrument or integrate your instrument into an established .NET based control framework.

- [Section 7.1](#) for help [Installing the LabOne .NET API](#).
- [Section 7.2](#) for help [Getting Started with the LabOne .NET API](#),
- [Section 7.3](#) for [LabOne .NET API Examples](#).

Note

This chapter and the examples are not intended to be a .NET and Visual Studio or an introduction to any specific programming language.

7.1. Installing the LabOne .NET API

7.1.1. Requirements

To use LabOne's .NET API, `ziDotNET`, a Microsoft Visual Studio installation is required. The .NET API is a class library supporting x64 and win32 platforms. As the API is platform specific the project also needs to be platform specific.

The LabOne .NET API `ziDotNET` is included in a standard LabOne installation. No installation as such is required, but the corresponding dynamically linked library (DLL) files need to be copied to the folder of the Visual Studio solution, and a few configuration steps must be performed. The main LabOne installer is available from Zurich Instruments' [download page](#).

7.2. Getting Started with the LabOne .NET API

This section introduces the user to the LabOne .NET API. In order to use the LabOne API for .NET applications two DLL libraries should be copied to the application execution folder. The libraries are platform specific. Therefore, the project platform of the project should be restricted either to x64 or win32 CPU architecture. The following figures illustrate the initial steps to create a C# project using the LabOne API. The setup for other languages like Visual Basic or F# is equivalent.

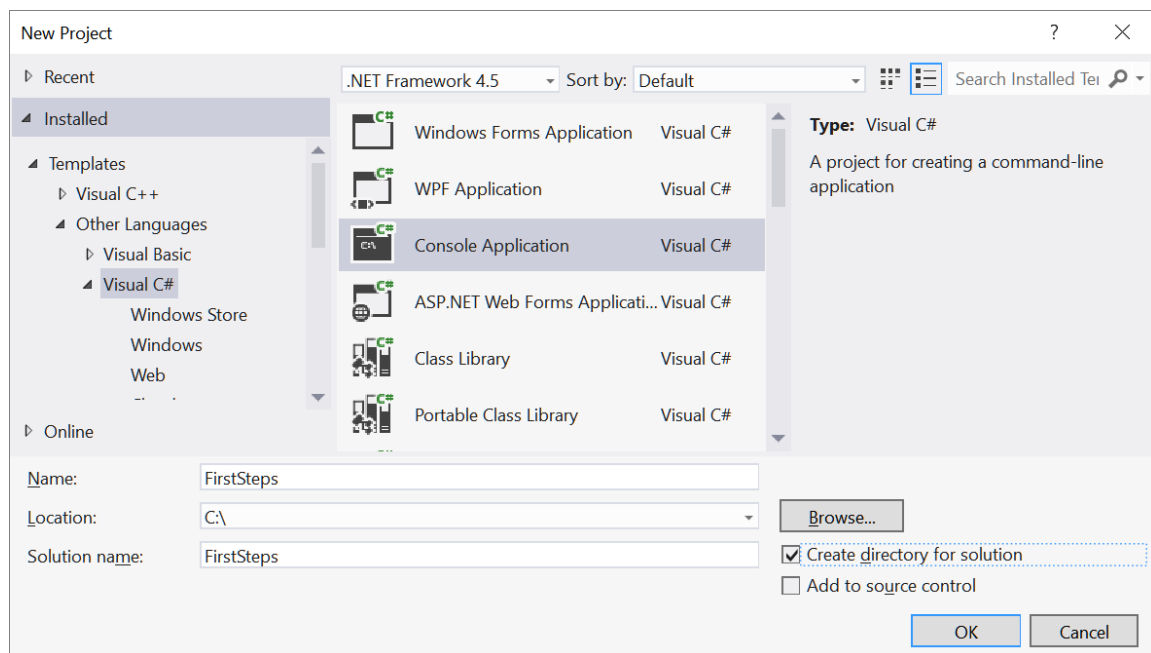


Figure 7.1. Creating a new C# project based on a solution.

Create a new project and choose Visual C# as a programming language and Console Application as target.

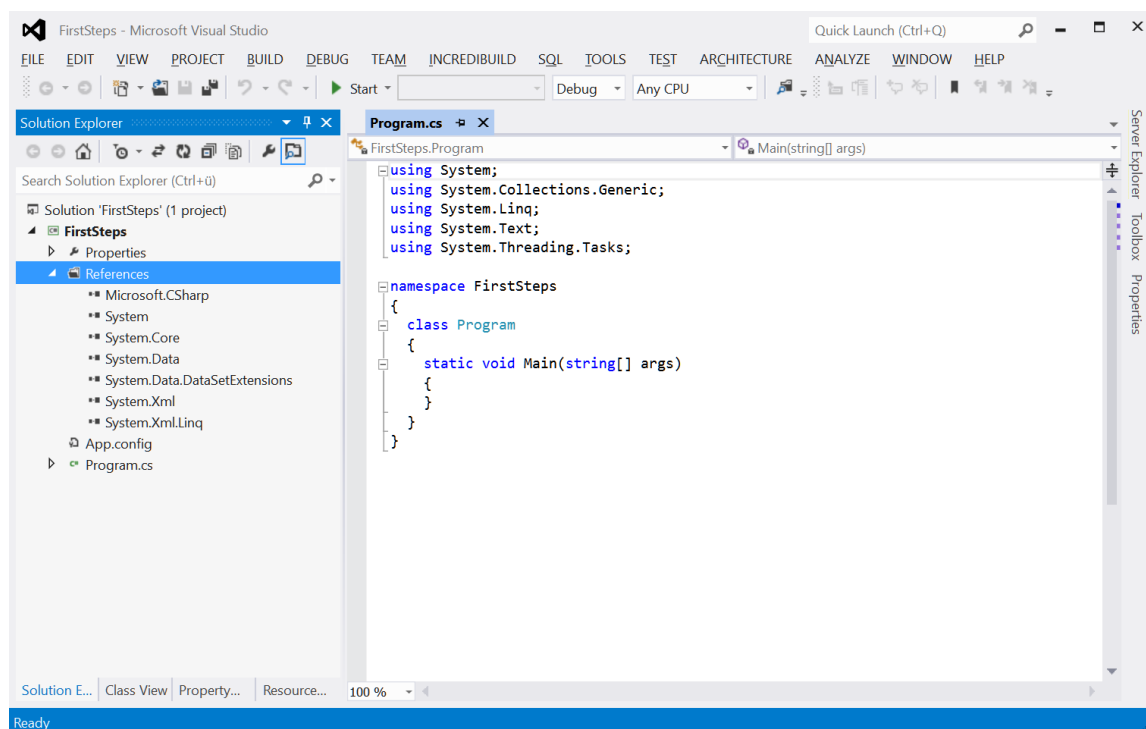


Figure 7.2. C# project with main program code opened in editor. Initially the project will support any CPU architecture. The ziDotNET API only supports x64 and win32 platforms.

The first step which needs to be done is to define the target platform as initially a Visual C# project is platform independent. To do this, click on the **Active solution platform** box, select **Configuration Manager...** to open the the **Configuration Manager**. In the following window click on the arrow under platform add a **New target** and choose **x64** (Figure 7.3).

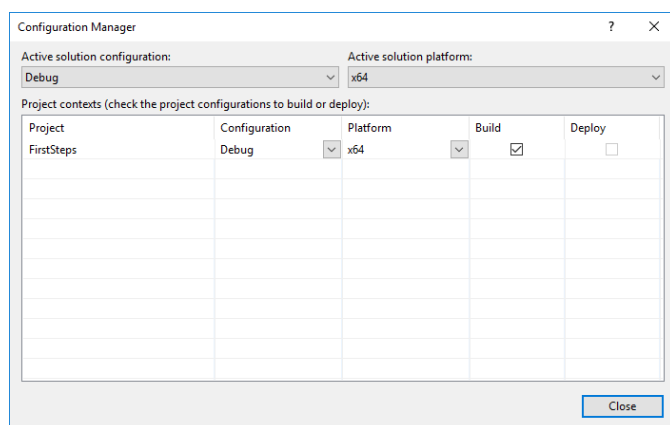


Figure 7.3. C# project with main program code opened in editor. Initially the project will support any CPU architecture. The ziDotNET API only supports x64 and win32 platforms.

The LabOne API for .NET consists of two DLLs for each platform that supply all functionality for connecting to the LabOne Data Servers on the specific platform (x64 and win32) and executing LabOne Modules. For simplicity we only discuss the x64 platform in this section, but the needed steps are analogous for the win32 platform. For x64 the two DLLs are ziDotNETCore-win64.dll and ziDotNET-win64.dll. The two DLL must accompany the executable using the functionality. The dll files are installed under your LabOne installation path in the API/DotNet folder (usually `C:\Program Files\Zurich Instruments\LabOne\API\DotNet`). Copy the two DLLs for your platform into the solution folder.

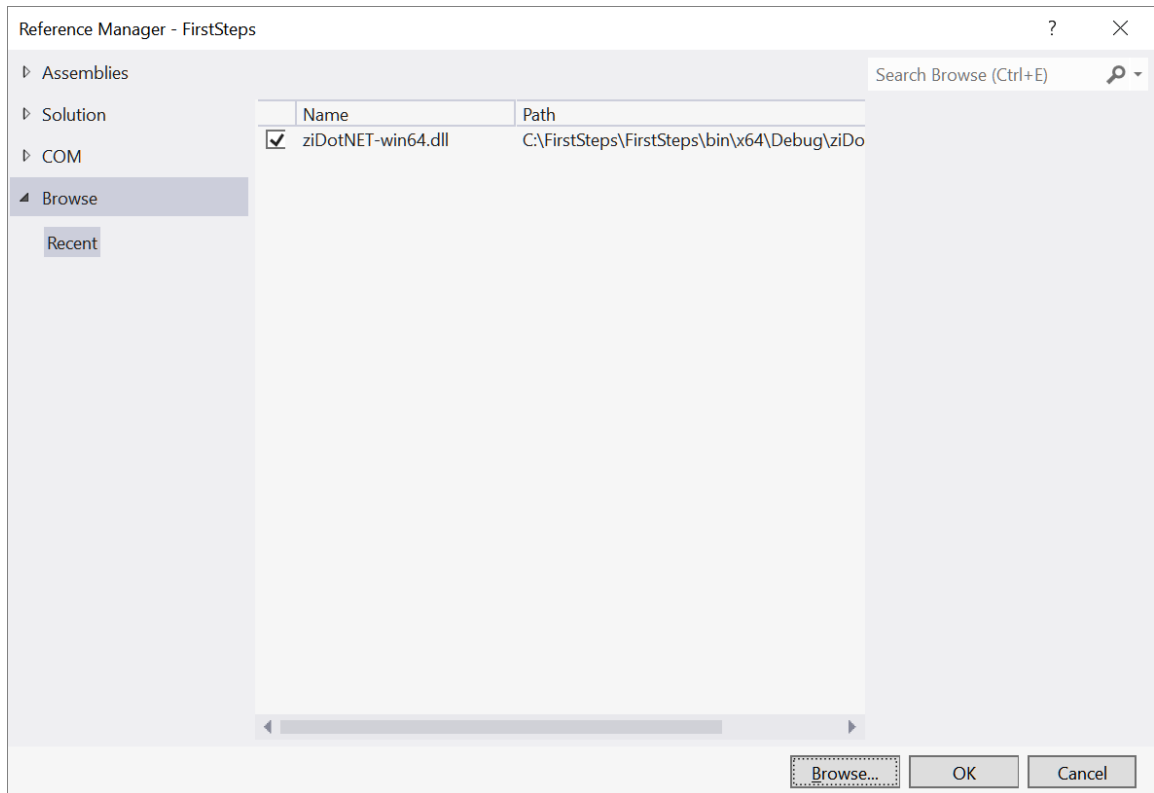


Figure 7.4. Reference to the API DLL ziDotNET for the specific platform.

To add the DLL to the project go to the solution explorer of your project (Figure 7.2) and right click on References and add the ziDotNET-win64.dll (Figure 7.4)

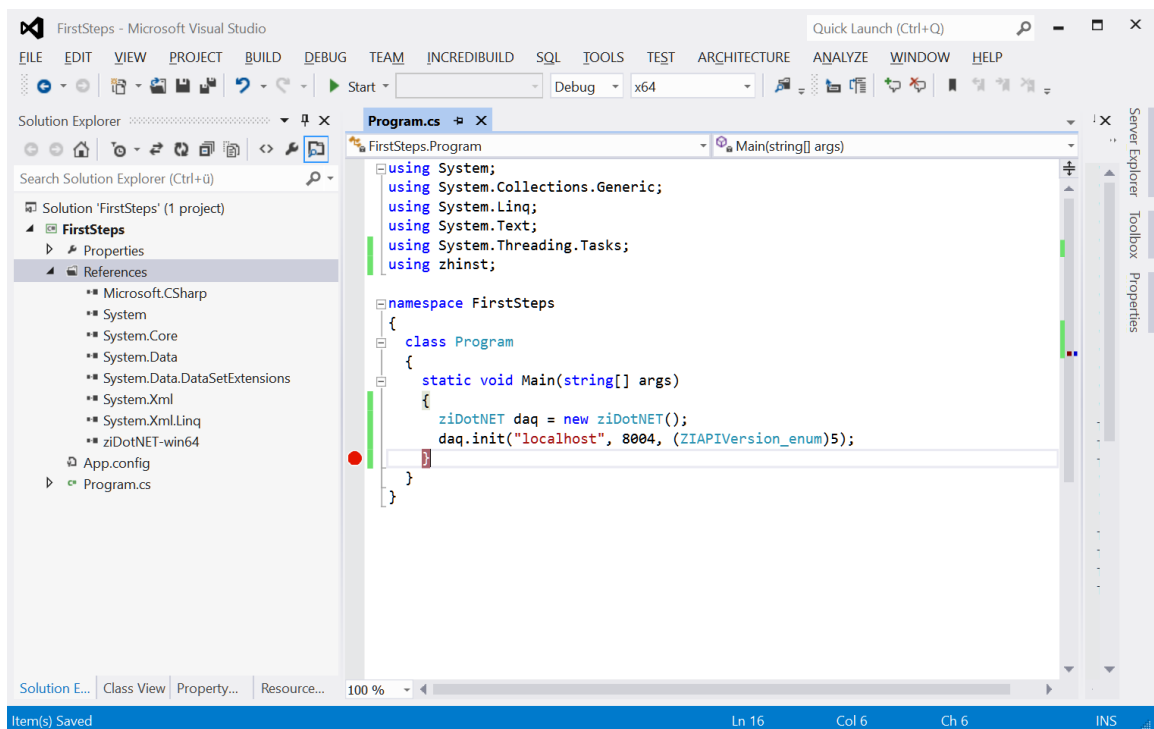


Figure 7.5. Added `using zhinst;` statement and code to open the connection to the server.

Figure 7.5 shows a first simple program which is done by adding `using zhinst;` to the include directive and the following code to the main body.

```
ziDotNET daq = new ziDotNET();  
daq.init("localhost", 8004, (ZIAPIVersion_enum) 5);
```

If everything is configured correctly, the code compiles and when executed opens a session to a running LabOne data server and closes it before exiting the program.

7.3. LabOne .NET API Examples

The source code for the following program (Examples.cs) can be found in your LabOne installation path in the API/DotNet folder (usually C:\Program Files\Zurich Instruments\LabOne\API\DotNET).

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;
using zhinst;
using System.IO;

namespace test_ziDotNET
{
    [TestClass]
    public class Examples
    {
        // device name needs to be adjusted to match the device in use
        static String dev = "dev3133";
        [ClassInitialize]
        public static void ClassSetup(TestContext a)
        {
            if (a.Properties.Contains("device"))
            {
                dev = a.Properties["device"].ToString();
            }
            System.Diagnostics.Trace.WriteLine("Using device " + dev);
        }

        // The resetDeviceToDefault will reset the device settings
        // to factory default. The call is quite expensive
        // in runtime. Never use it inside loops!
        private void resetDeviceToDefault(ziDotNET daq)
        {
            // The HF2 devices do not support the preset functionality.
            if (isDeviceFamily(daq, "HF2"))
            {
                daq.setDouble(String.Format("/{0}/demods/*/rate", dev), 250);
            }
            else
            {
                daq.setInt(String.Format("/{0}/system/preset/index", dev), 0);
                daq.setInt(String.Format("/{0}/system/preset/load", dev), 1);
                while (daq.getInt(String.Format("/{0}/system/preset/busy", dev)) != 0) {
                    System.Threading.Thread.Sleep(100);
                }
                System.Threading.Thread.Sleep(1000);
            }
        }

        // The isDeviceFamily checks for a specific device family.
        // Currently available families: "HF2", "UHF", "MF"
        private bool isDeviceFamily(ziDotNET daq, String family)
        {
            String path = String.Format("/{0}/features/devtype", dev);
            String devType = daq.getBytes(path);
            return devType.StartsWith(family);
        }

        // The hasOption function checks if the device
        // does support a specific functionality, thus
        // has installed the option.
        private bool hasOption(ziDotNET daq, String option)
```

```
{
    String path = String.Format("/{0}/features/options", dev);
    String options = daq.getBytes(path);
    return options.Contains(option);
}

// Please handle version mismatches depending on your
// application requirements. Version mismatches often relate
// to functionality changes of some nodes. The API interface is still
// identical. We strongly recommend to keep the version of the
// API and data server identical. Following approaches are possible:
// - Convert version mismatch to a warning for the user to upgrade / downgrade
// - Convert version mismatch to an error to enforce full matching
// - Do an automatic upgrade / downgrade
private void apiServerVersionCheck(ziDotNET daq)
{
    String serverVersion = daq.getBytes("/zi/about/version");
    String apiVersion = daq.version();

    Assert.AreEqual(serverVersion, apiVersion,
        "Version mismatch between LabOne API and Data Server.");
}

// Connect initializes a session on the server.
private ziDotNET connect()
{
    ziDotNET daq = new ziDotNET();
    String id = daq.discoveryFind(dev);
    String iface = daq.discoveryGetValueS(dev, "connected");
    if (string.IsNullOrEmpty(iface))
    {
        // Device is not connected to the server
        String ifacesList = daq.discoveryGetValueS(dev, "interfaces");
        // Select the first available interface and use it to connect
        string[] ifaces = ifacesList.Split('\n');
        if (ifaces.Length > 0)
        {
            iface = ifaces[0];
        }
    }
    String host = daq.discoveryGetValueS(dev, "serveraddress");
    long port = daq.discoveryGetValueI(dev, "serverport");
    long api = daq.discoveryGetValueI(dev, "apilevel");
    System.Diagnostics.Trace.WriteLine(
        String.Format("Connecting to server {0}:{1} with API level {2}",
            host, port, api));
    daq.init(host, Convert.ToInt16(port), (ZIAPIVersion_enum)api);
    // Ensure that LabOne API and LabOne Data Server are from
    // the same release version.
    apiServerVersionCheck(daq);
    // If device is not yet connected a reconnect
    // will not harm.
    System.Diagnostics.Trace.WriteLine(
        String.Format("Connecting to {0} on interface {1}", dev, iface));
    daq.connectDevice(dev, iface, "");

    return daq;
}

// ExamplePollDemodSample connects to the device,
// subscribes to a demodulator, polls the data for 0.1 s
// and returns the data.
[TestMethod]
public void ExamplePollDemodSample()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
}
```

```
String path = String.Format("/{0}/demods/0/sample", dev);
daq.subscribe(path);
// After the subscribe the poll can be executed
// continuously within a loop
Lookup lookup = daq.poll(0.1, 100, 0, 1);
Dictionary<String, Chunk[]> nodes = lookup.nodes; // Iterable nodes
Chunk[] chunks = lookup[path]; // Iterable chunks
Chunk chunk = lookup[path][0]; // Single chunk
// Vector of samples
ZIDemodSample[] demodSamples = lookup[path][0].demodSamples;
// Single sample
ZIDemodSample demodSample0 = lookup[path][0].demodSamples[0];
daq.disconnect();

Assert.AreNotEqual(0, demodSample0.timeStamp);
}

// ExamplePollImpedanceSample connects to the device,
// subscribes to a impedance stream, polls the data for 0.1 s
// and returns the data.
[TestMethod]
public void ExamplePollImpedanceSample()
{
    ziDotNET daq = connect();
    // This example only works for devices with installed
    // Impedance Analyzer (IA) option.
    if (!hasOption(daq, "IA"))
    {
        daq.disconnect();
        return;
    }
    resetDeviceToDefault(daq);
    // Enable impedance control
    daq.setInt(String.Format("/{0}/imps/0/enable", dev), 1);
    // Return R and Cp
    daq.setInt(String.Format("/{0}/imps/0/model", dev), 0);
    // Enable user compensation
    daq.setInt(String.Format("/{0}/imps/0/calib/user/enable", dev), 1);
    // Wait until auto ranging has settled
    System.Threading.Thread.Sleep(4000);
    // Subscribe to the impedance data stream
    String path = String.Format("/{0}/imps/0/sample", dev);
    daq.subscribe(path);
    // After the subscribe the poll can be executed
    // continuously within a loop
    Lookup lookup = daq.poll(0.1, 100, 0, 1);
    Dictionary<String, Chunk[]> nodes = lookup.nodes; // Iterable nodes
    Chunk[] chunks = lookup[path]; // Iterable chunks
    Chunk chunk = lookup[path][0]; // Single chunk
    // Vector of samples
    ZIImpedanceSample[] impedanceSamples = lookup[path][0].impedanceSamples;
    // Single sample
    ZIImpedanceSample impedanceSample0 = lookup[path][0].impedanceSamples[0];
    // Extract the R||C representation values
    System.Diagnostics.Trace.WriteLine(
        String.Format("Impedance Resistor value: {0} Ohm.",
            impedanceSample0.param0));
    System.Diagnostics.Trace.WriteLine(
        String.Format("Impedance Capacitor value: {0} F.",
            impedanceSample0.param1));
    daq.disconnect();

    Assert.AreNotEqual(0, impedanceSample0.timeStamp);
}

// ExamplePollDoubleData is similar to ExamplePollDemodSample,
// but it subscribes and polls floating point data.
```

```
[TestMethod]
public void ExamplePollDoubleData()
{
    ziDotNET daq = connect();
    String path = String.Format("/{0}/auxouts/0/value", dev);
    daq.getAsEvent(path);
    daq.subscribe(path);
    Lookup lookup = daq.poll(1, 100, 0, 1);
    Dictionary<String, Chunk[]> nodes = lookup.nodes; // Iterable nodes
    Chunk[] chunks = lookup[path]; // Iterable chunks
    Chunk chunk = lookup[path][0]; // Single chunk
    ZIDoubleData[] doubleData = lookup[path][0].doubleData; // Vector of samples
    ZIDoubleData doubleData0 = lookup[path][0].doubleData[0]; // Single sample
    daq.disconnect();

    Assert.AreNotEqual(0, doubleData0.timeStamp);
}

// ExamplePollPwaData is similar to ExamplePollDemodSample,
// but it subscribes and polls periodic waveform analyzer
// data from a device with the Boxcar option.
[TestMethod, Timeout(10000)]
public void ExamplePollPwaData()
{
    ziDotNET daq = connect();
    // The PWA example only works for devices with installed Boxcar (BOX) option
    if (hasOption(daq, "BOX"))
    {
        String enablePath = String.Format("/{0}/inputpwas/0/enable", dev);
        daq.setInt(enablePath, 1);
        String path = String.Format("/{0}/inputpwas/0/wave", dev);
        daq.subscribe(path);
        Lookup lookup = daq.poll(1, 100, 0, 1);
        UInt64 timeStamp = lookup[path][0].pwaWaves[0].timeStamp;
        UInt64 sampleCount = lookup[path][0].pwaWaves[0].sampleCount;
        UInt32 inputSelect = lookup[path][0].pwaWaves[0].inputSelect;
        UInt32 oscSelect = lookup[path][0].pwaWaves[0].oscSelect;
        UInt32 harmonic = lookup[path][0].pwaWaves[0].harmonic;
        Double frequency = lookup[path][0].pwaWaves[0].frequency;
        Byte type = lookup[path][0].pwaWaves[0].type;
        Byte mode = lookup[path][0].pwaWaves[0].mode;
        Byte overflow = lookup[path][0].pwaWaves[0].overflow;
        Byte commensurable = lookup[path][0].pwaWaves[0].commensurable;
        double[] grid = lookup[path][0].pwaWaves[0].binPhase;
        double[] x = lookup[path][0].pwaWaves[0].x;
        double[] y = lookup[path][0].pwaWaves[0].y;
        String fileName = Environment.CurrentDirectory + "/pwa.txt";
        System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
        file.WriteLine("TimeStamp: {0}", timeStamp);
        file.WriteLine("Sample Count: {0}", sampleCount);
        file.WriteLine("Input Select: {0}", inputSelect);
        file.WriteLine("Osc Select: {0}", oscSelect);
        file.WriteLine("Frequency: {0}", frequency);
        for (int i = 0; i < grid.Length; ++i)
        {
            file.WriteLine("{0} {1} {2}", grid[i], x[i], y[i]);
        }
        file.Close();

        Assert.AreNotEqual(0, timeStamp);
        Assert.AreNotEqual(0, sampleCount);
        Assert.AreNotEqual(0, grid.Length);
    }
    daq.disconnect();
}

// ExamplePollScopeData is similar to ExamplePollDemodSample,
```

```
// but it subscribes and polls scope data.
[TestMethod]
public void ExamplePollScopeData()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);

    String enablePath = String.Format("/{0}/scopes/0/enable", dev);
    daq.setInt(enablePath, 1);
    String path = String.Format("/{0}/scopes/0/wave", dev);
    daq.subscribe(path);
    Lookup lookup = daq.poll(1, 100, 0, 1);
    UInt64 timeStamp = lookup[path][0].scopeWaves[0].header.timeStamp;
    UInt64 sampleCount = lookup[path][0].scopeWaves[0].header.totalSamples;
    daq.disconnect();

    Assert.AreNotEqual(0, timeStamp);
    Assert.AreNotEqual(0, sampleCount);
}

// ExampleGetDemodSample reads the demodulator sample value of the specified
node.
[TestMethod]
public void ExampleGetDemodSample()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
    String path = String.Format("/{0}/demods/0/sample", dev);
    ZIDemodSample sample = daq.getDemodSample(path);
    System.Diagnostics.Trace.WriteLine(sample.frequency, "Sample frequency");
    daq.disconnect();

    Assert.AreNotEqual(0, sample.timeStamp);
}

// ExampleSweeper instantiates a sweeper module and executes a sweep
// over 100 data points from 1kHz to 100kHz and writes the result into a file.
[TestMethod, Timeout(40000)]
public void ExampleSweeper()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
    ziModule sweep = daq.sweeper();
    sweep.setByte("sweep/device", dev);
    sweep.setDouble("sweep/start", 1e3);
    sweep.setDouble("sweep/stop", 1e5);
    sweep.setDouble("sweep/samplecount", 100);
    String path = String.Format("/{0}/demods/0/sample", dev);
    sweep.subscribe(path);
    sweep.execute();
    while (!sweep.finished())
    {
        System.Threading.Thread.Sleep(100);
        double progress = sweep.progress() * 100;
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
    }
    Lookup lookup = sweep.read();
    double[] grid = lookup[path][0].sweeperDemodWaves[0].grid;
    double[] x = lookup[path][0].sweeperDemodWaves[0].x;
    double[] y = lookup[path][0].sweeperDemodWaves[0].y;
    String fileName = Environment.CurrentDirectory + "/sweep.txt";
    System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
    ZIChunkHeader header = lookup[path][0].header;
    // Raw system time is the number of microseconds since linux epoch
    file.WriteLine("Raw System Time: {0}", header.systemTime);
    // Use the utility function ziSystemTimeToDateTime to convert to DateTime
of .NET
```



```
        file.WriteLine("Converted System Time: {0}",
            ziUtility.ziSystemTimeToDateTime(lookup[path][0].header.systemTime));
        file.WriteLine("Created Timestamp: {0}", header.createdTimeStamp);
        file.WriteLine("Changed Timestamp: {0}", header.changedTimeStamp);
        for (int i = 0; i < grid.Length; ++i)
        {
            file.WriteLine("{0} {1} {2}", grid[i], x[i], y[i]);
        }
        file.Close();

        Assert.AreEqual(1.0, sweep.progress());
        Assert.AreNotEqual(0, grid.Length);

        daq.disconnect();
    }

    // ExampleImpedanceSweeper instantiates a sweeper module and prepares
    // all settings for an impedance sweep over 30 data points.
    // The results are written to a file.
    [TestMethod, Timeout(40000)]
    public void ExampleImpedanceSweeper()
    {
        ziDotNET daq = connect();
        // This example only works for devices with installed
        // Impedance Analyzer (IA) option.
        if (!hasOption(daq, "IA"))
        {
            daq.disconnect();
            return;
        }

        resetDeviceToDefault(daq);
        // Enable impedance control
        daq.setInt(String.Format("/{0}/imps/0/enable", dev), 1);
        // Return D and Cs
        daq.setInt(String.Format("/{0}/imps/0/model", dev), 4);
        // Enable user compensation
        daq.setInt(String.Format("/{0}/imps/0/calib/user/enable", dev), 1);

        // ensure correct settings of order and oscselect
        daq.setInt(String.Format("/{0}/imps/0/demod/order", dev), 8);
        daq.setInt(String.Format("/{0}/imps/0/demod/oscselect", dev), 0);
        daq.sync();

        ziModule sweep = daq.sweeper();
        // Sweeper settings
        sweep.setByte("sweep/device", dev);
        sweep.setDouble("sweep/start", 1e3);
        sweep.setDouble("sweep/stop", 5e6);
        sweep.setDouble("sweep/samplecount", 30);
        sweep.setDouble("sweep/order", 8);
        sweep.setDouble("sweep/settling/inaccuracy", 0.0100000);
        sweep.setDouble("sweep/bandwidthcontrol", 2);
        sweep.setDouble("sweep/maxbandwidth", 10.0);
        sweep.setDouble("sweep/bandwidthoverlap", 1);
        sweep.setDouble("sweep/xmapping", 1);
        sweep.setDouble("sweep/omegasuppression", 100.0);
        sweep.setDouble("sweep/averaging/sample", 200);
        sweep.setDouble("sweep/averaging/time", 0.100);
        sweep.setDouble("sweep/averaging/tc", 20.0);
        String path = String.Format("/{0}/imps/0/sample", dev);
        sweep.subscribe(path);
        sweep.execute();
        while (!sweep.finished())
        {
            System.Threading.Thread.Sleep(100);
            double progress = sweep.progress() * 100;
```

```
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
    }
    Lookup lookup = sweep.read();
    double[] grid = lookup[path][0].sweeperImpedanceWaves[0].grid;
    double[] x = lookup[path][0].sweeperImpedanceWaves[0].realz;
    double[] y = lookup[path][0].sweeperImpedanceWaves[0].imagz;
    double[] param0 = lookup[path][0].sweeperImpedanceWaves[0].param0;
    double[] param1 = lookup[path][0].sweeperImpedanceWaves[0].param1;
    // Save measurement data to file
    String fileName = Environment.CurrentDirectory + "/impedance.txt";
    System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
    ZIChunkHeader header = lookup[path][0].header;
    // Raw system time is the number of microseconds since linux epoch
    file.WriteLine("Raw System Time: {0}", header.systemTime);
    // Use the utility function ziSystemTimeToDateTime to convert to DateTime
of .NET
    file.WriteLine("Converted System Time: {0}",
ziUtility.ziSystemTimeToDateTime(lookup[path][0].header.systemTime));
    file.WriteLine("Created Timestamp: {0}", header.createdTimeStamp);
    file.WriteLine("Changed Timestamp: {0}", header.changedTimeStamp);
    for (int i = 0; i < grid.Length; ++i)
    {
        file.WriteLine("{0} {1} {2} {3} {4}",
            grid[i],
            x[i],
            y[i],
            param0[i],
            param1[i]);
    }
    file.Close();

    Assert.AreEqual(1.0, sweep.progress());
    Assert.AreNotEqual(0, grid.Length);
}

// ExampleImpedanceCompensation does a user compensation
// of the impedance analyser.

[TestMethod, Timeout(30000)]
public void ExampleImpedanceCompensation()
{
    ziDotNET daq = connect();
    // This example only works for devices with installed
    // Impedance Analyzer (IA) option.
    if (!hasOption(daq, "IA"))
    {
        daq.disconnect();
        return;
    }

    resetDeviceToDefault(daq);

    // Enable impedance control
    daq.setInt(String.Format("/{0}/imps/0/enable", dev), 1);
    ziModule calib = daq.impedanceModule();
    calib.execute();
    calib.setByte("impedanceModule/device", dev);
    System.Threading.Thread.Sleep(200);
    calib.setInt("impedanceModule/mode", 4);
    calib.setDouble("impedanceModule/loads/2/r", 1000.0);
    calib.setDouble("impedanceModule/loads/2/c", 0.0);
    calib.setDouble("impedanceModule/freq/start", 100.0);
    calib.setDouble("impedanceModule/freq/stop", 500e3);
    calib.setDouble("impedanceModule/freq/samplecount", 21);

    daq.setInt(String.Format("/{0}/imps/0/demod/order", dev), 8);
    daq.setInt(String.Format("/{0}/imps/0/demod/oscselect", dev), 0);
```

```
    daq.sync();

    calib.setInt("impedanceModule/step", 2);
    calib.setInt("impedanceModule/calibrate", 1);
    while (true)
    {
        System.Threading.Thread.Sleep(100);
        double progress = calib.progress() * 100;
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
        Int64 calibrate = calib.getInt("impedanceModule/calibrate");
        if (calibrate == 0)
        {
            break;
        }
    }
    String message = calib.getString("impedanceModule/message");
    System.Diagnostics.Trace.WriteLine(message, "Message");
    Assert.AreEqual(0, calib.progress());
}

// ExampleSpectrum instantiates the spectrum module,
// reads the data and writes the result in to a file.
[TestMethod, Timeout(20000)]
public void ExampleSpectrum()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
    ziModule spectrum = daq.spectrum();
    spectrum.setByte("zoomFFT/device", dev);
    spectrum.setInt("zoomFFT/bit", 10);
    String path = String.Format("/{0}/demods/0/sample", dev);
    spectrum.subscribe(path);
    spectrum.execute();
    while (!spectrum.finished())
    {
        System.Threading.Thread.Sleep(100);
        double progress = spectrum.progress() * 100;
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
    }
    Lookup lookup = spectrum.read();
    double[] grid = lookup[path][0].spectrumWaves[0].grid;
    double[] x = lookup[path][0].spectrumWaves[0].x;
    double[] y = lookup[path][0].spectrumWaves[0].y;
    String fileName = Environment.CurrentDirectory + "/spectrum.txt";
    System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
    for (int i = 0; i < grid.Length; ++i)
    {
        file.WriteLine("{0} {1} {2}", grid[i], x[i], y[i]);
    }
    file.Close();

    Assert.AreEqual(1.0, spectrum.progress());
    Assert.AreEqual(0, grid.Length);

    daq.disconnect();
}

// ExampleSwTrigger uses the software trigger to record data
// and writes the result in to a file.
[TestMethod, Timeout(20000)]
public void ExampleSwTrigger()
{
    ziDotNET daq = connect();
    if ((isDeviceFamily(daq, "MF") && !hasOption(daq, "MD")) ||
        (isDeviceFamily(daq, "HF2") && !hasOption(daq, "MFK")))
    {
```

```
        daq.disconnect();
        return;
    }

    resetDeviceToDefault(daq);
    daq.setInt(String.Format("/{0}/demods/0/oscsselect", dev), 0);
    daq.setInt(String.Format("/{0}/demods/1/oscsselect", dev), 1);
    daq.setDouble(String.Format("/{0}/oscs/0/freq", dev), 2e6);
    daq.setDouble(String.Format("/{0}/oscs/1/freq", dev), 2.0001e6);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/*", dev), 0);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/0", dev), 1);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/1", dev), 1);
    daq.setInt(String.Format("/{0}/sigouts/0/on", dev), 1);
    daq.setDouble(String.Format("/{0}/sigouts/0/amplitudes/0", dev), 0.2);
    daq.setDouble(String.Format("/{0}/sigouts/0/amplitudes/1", dev), 0.2);
    ziModule trigger = daq.swTrigger();
    trigger.setByte("trigger/device", dev);
    trigger.setInt("trigger/0/type", 1);
    trigger.setDouble("trigger/0/level", 0.1);
    trigger.setDouble("trigger/0/hysteresis", 0.01);
    trigger.setDouble("trigger/0/bandwidth", 0.0);
    String path = String.Format("/{0}/demods/0/sample", dev);
    trigger.subscribe(path);
    String triggerPath = String.Format("/{0}/demods/0/sample.R", dev);
    trigger.setByte("trigger/0/triggernode", triggerPath);
    trigger.execute();
    while (!trigger.finished())
    {
        System.Threading.Thread.Sleep(100);
        double progress = trigger.progress() * 100;
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
    }
    Lookup lookup = trigger.read();
    ZIDemodSample[] demodSample = lookup[path][0].demodSamples;
    String fileName = Environment.CurrentDirectory + "/swtrigger.txt";
    System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
    ZIChunkHeader header = lookup[path][0].header;
    // Raw system time is the number of microseconds since linux epoch
    file.WriteLine("Raw System Time: {0}", header.systemTime);
    // Use the utility function ziSystemTimeToDateTime to convert to DateTime
of .NET
    file.WriteLine("Converted System Time: {0}",
        ziUtility.ziSystemTimeToDateTime(lookup[path][0].header.systemTime));
    file.WriteLine("Created Timestamp: {0}", header.createdTimeStamp);
    file.WriteLine("Changed Timestamp: {0}", header.changedTimeStamp);
    file.WriteLine("Flags: {0}", header.flags);
    file.WriteLine("Name: {0}", header.name);
    file.WriteLine("Status: {0}", header.status);
    file.WriteLine("Group Index: {0}", header.groupIndex);
    file.WriteLine("Color: {0}", header.color);
    file.WriteLine("Active Row: {0}", header.activeRow);
    file.WriteLine("Trigger Number: {0}", header.triggerNumber);
    file.WriteLine("Grid Rows: {0}", header.gridRows);
    file.WriteLine("Grid Cols: {0}", header.gridCols);
    file.WriteLine("Grid Mode: {0}", header.gridMode);
    file.WriteLine("Grid Operation: {0}", header.gridOperation);
    file.WriteLine("Grid Direction: {0}", header.gridDirection);
    file.WriteLine("Grid Repetitions: {0}", header.gridRepetitions);
    file.WriteLine("Grid Col Delta: {0}", header.gridColDelta);
    file.WriteLine("Grid Col Offset: {0}", header.gridColOffset);
    file.WriteLine("Bandwidth: {0}", header.bandwidth);
    file.WriteLine("Center: {0}", header.center);
    file.WriteLine("NENBW: {0}", header.nenbw);
    for (int i = 0; i < demodSample.Length; ++i)
    {
        file.WriteLine("{0} {1} {2}",
            demodSample[i].frequency,
```

```
        demodSample[i].x,
        demodSample[i].y);
    }
    file.Close();

    Assert.AreEqual(1, trigger.progress());
    Assert.AreNotEqual(0, demodSample.Length);

    daq.disconnect();
}

// ExampleScopeModule instantiates a scope module.
[TestMethod, Timeout(20000)]
public void ExampleScopeModule()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
    ziModule scopeModule = daq.scopeModule();
    String path = String.Format("/{0}/scopes/0/wave", dev);
    scopeModule.subscribe(path);
    scopeModule.execute();
    // The HF2 devices do not have a single event functionality.
    if (!isDeviceFamily(daq, "HF2"))
    {
        daq.setInt(String.Format("/{0}/scopes/0/single", dev), 1);
        daq.setInt(String.Format("/{0}/scopes/0/trigenable", dev), 0);
    }
    daq.setInt(String.Format("/{0}/scopes/0/enable", dev), 1);

    Lookup lookup;
    bool allSegments = false;
    do
    {
        System.Threading.Thread.Sleep(100);
        double progress = scopeModule.progress() * 100;
        System.Diagnostics.Trace.WriteLine(progress, "Progress");
        lookup = scopeModule.read();
        if (lookup.nodes.ContainsKey(path))
        {
            ZIScopeWave[] scopeWaves = lookup[path][0].scopeWaves;
            UInt64 totalSegments = scopeWaves[0].header.totalSegments;
            UInt64 segmentNumber = scopeWaves[0].header.segmentNumber;
            allSegments = (totalSegments == 0) ||
                (segmentNumber >= totalSegments - 1);
        }
    } while (!allSegments);
    ZIScopeWave[] scopeWaves1 = lookup[path][0].scopeWaves;
    float[,] wave = SimpleValue.getFloatVec2D(scopeWaves1[0].wave);
    // ...
    System.Diagnostics.Trace.WriteLine(wave.Length, "Wave Size");
    Assert.AreNotEqual(0, wave.Length);
    daq.disconnect();
}

// ExampleDeviceSettings instantiates a deviceSettings module and performs a save
// and load of device settings. The LabOne UI uses this module to save and
// load the device settings.
[TestMethod, Timeout(15000)]
public void ExampleDeviceSettings()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);
    ziModule settings = daq.deviceSettings();
    // First save the current device settings
    settings.setString("deviceSettings/device", dev);
    settings.setString("deviceSettings/command", "save");
    settings.setString("deviceSettings/filename", "test_settings");
}
```

```
settings.setString("deviceSettings/path", Environment.CurrentDirectory);
settings.execute();
while (!settings.finished())
{
    System.Threading.Thread.Sleep(100);
}
// Remember the current device parameter for later comparison
String path = String.Format("/{0}/oscs/0/freq", dev);
Double originalValue = daq.getDouble(path);
// Change the parameter
daq.setDouble(path, 2 * originalValue);
// Load device settings from file
settings.setString("deviceSettings/device", dev);
settings.setString("deviceSettings/command", "load");
settings.setString("deviceSettings/filename", "test_settings");
settings.setString("deviceSettings/path", Environment.CurrentDirectory);
settings.execute();
while (!settings.finished())
{
    System.Threading.Thread.Sleep(100);
}
// Check the restored parameter
Double newValue = daq.getDouble(path);

Assert.AreEqual(originalValue, newValue);

daq.disconnect();
}

// ExamplePidAdvisor shows the usage of the PID advisor
[TestMethod, Timeout(40000)]
public void ExamplePidAdvisor()
{
    ziDotNET daq = connect();
    if (!hasOption(daq, "PID"))
    {
        daq.disconnect();
        return;
    }

    resetDeviceToDefault(daq);

    daq.setInt(String.Format("/{0}/demods/*/rate", dev), 0);
    daq.setInt(String.Format("/{0}/demods/*/trigger", dev), 0);
    daq.setInt(String.Format("/{0}/sigouts/*/enables/*", dev), 0);
    daq.setInt(String.Format("/{0}/demods/*/enable", dev), 0);
    daq.setInt(String.Format("/{0}/scopes/*/enable", dev), 0);

    // now the settings relevant to this experiment
    // PID configuration.
    double target_bw = 10e3;    // Target bandwidth (Hz).
    int pid_input = 3;          // PID input (3 = Demod phase).
    int pid_input_channel = 0;   // Demodulator number.
    double setpoint = 0.0;      // Phase setpoint.
    int phase_unwrap = 1;       //
    int pid_output = 2;         // PID output (2 = oscillator frequency).
    int pid_output_channel = 0;  // The index of the oscillator controlled by PID.
    double pid_center_frequency = 500e3; // (Hz).
    double pid_limits = 10e3;   // (Hz).

    if (!isDeviceFamily(daq, "HF2"))
    {
        daq.setInt(String.Format("/{0}/pids/0/input", dev), pid_input);
        daq.setInt(String.Format("/{0}/pids/0/inputchannel", dev),
pid_input_channel);
        daq.setDouble(String.Format("/{0}/pids/0/setpoint", dev), setpoint);
    }
}
```

```
        daq.setInt(String.Format("/{0}/pids/0/output", dev), pid_output);
        daq.setInt(String.Format("/{0}/pids/0/outputchannel", dev),
pid_output_channel);
        daq.setDouble(String.Format("/{0}/pids/0/center", dev),
pid_center_frequency);
        daq.setInt(String.Format("/{0}/pids/0/enable", dev), 0);
        daq.setInt(String.Format("/{0}/pids/0/phaseunwrap", dev), phase_unwrap);
        daq.setDouble(String.Format("/{0}/pids/0/limitlower", dev), -pid_limits);
        daq.setDouble(String.Format("/{0}/pids/0/limitupper", dev), pid_limits);
    }
    // Perform a global synchronisation between the device and the data server:
    // Ensure that the settings have taken effect on the device before starting
    // the pidAdvisor.
    daq.sync();

    // set up PID Advisor
    ziModule pidAdvisor = daq.pidAdvisor();

    // Turn off auto-calc on param change. Enabled
    // auto calculation can be used to automatically
    // update response data based on user input.
    pidAdvisor.setInt("pidAdvisor/auto", 0);
    pidAdvisor.setByte("pidAdvisor/device", dev);
    pidAdvisor.setDouble("pidAdvisor/pid/targetbw", target_bw);

    // PID advising mode (bit coded)
    // bit 0: optimize/tune P
    // bit 1: optimize/tune I
    // bit 2: optimize/tune D
    // Example: mode = 7: Optimize/tune PID
    pidAdvisor.setInt("pidAdvisor/pid/mode", 7);

    // PID index to use (first PID of device: 0)
    pidAdvisor.setInt("pidAdvisor/index", 0);

    // DUT model
    // source = 1: Lowpass first order
    // source = 2: Lowpass second order
    // source = 3: Resonator frequency
    // source = 4: Internal PLL
    // source = 5: VCO
    // source = 6: Resonator amplitude
    pidAdvisor.setInt("pidAdvisor/dut/source", 4);

    if (isDeviceFamily(daq, "HF2"))
    {
        // Since the PLL and PID are 2 separate hardware units on the
        // device, we need to additionally specify that the PID
        // Advisor should model the HF2's PLL.
        pidAdvisor.setByte("pidAdvisor/pid/type", "pll");
    }

    // IO Delay of the feedback system describing the earliest response
    // for a step change. This parameter does not affect the shape of
    // the DUT transfer function
    pidAdvisor.setDouble("pidAdvisor/dut/delay", 0.0);

    // Other DUT parameters (not required for the internal PLL model)
    // pidAdvisor.setDouble('pidAdvisor/dut/gain', 1.0)
    // pidAdvisor.setDouble('pidAdvisor/dut/bw', 1000)
    // pidAdvisor.setDouble('pidAdvisor/dut/fcenter', 15e6)
    // pidAdvisor.setDouble('pidAdvisor/dut/damping', 0.1)
    // pidAdvisor.setDouble('pidAdvisor/dut/q', 10e3)

    // Start values for the PID optimization. Zero
    // values will imitate a guess. Other values can be
    // used as hints for the optimization process.
```

```
pidAdvisor.setDouble("pidAdvisor/pid/p", 0);
pidAdvisor.setDouble("pidAdvisor/pid/i", 0);
pidAdvisor.setDouble("pidAdvisor/pid/d", 0);
pidAdvisor.setInt("pidAdvisor/calculate", 0);

// Start the module thread
pidAdvisor.execute();
System.Threading.Thread.Sleep(1000);

// Advise
pidAdvisor.setInt("pidAdvisor/calculate", 1);
System.Diagnostics.Trace.WriteLine(
    "Starting advising. Optimization process may run up to a minute...");

var watch = System.Diagnostics.Stopwatch.StartNew();
while (true)
{
    double progress = pidAdvisor.progress() * 100;
    System.Diagnostics.Trace.WriteLine(progress, "Progress");
    System.Threading.Thread.Sleep(1000);
    Int64 calc = pidAdvisor.getInt("pidAdvisor/calculate");
    if (calc == 0)
    {
        break;
    }
}

watch.Stop();
var elapsedMs = watch.ElapsedMilliseconds;

System.Diagnostics.Trace.WriteLine(
    String.Format("Advice took {0} s.", watch.ElapsedMilliseconds / 1000.0));

// Get the advised values
double p_adv = pidAdvisor.getDouble("pidAdvisor/pid/p");
double i_adv = pidAdvisor.getDouble("pidAdvisor/pid/i");
double d_adv = pidAdvisor.getDouble("pidAdvisor/pid/d");
double dlimittimeconstant_adv =
    pidAdvisor.getDouble("pidAdvisor/pid/dlimittimeconstant");
double rate_adv = pidAdvisor.getDouble("pidAdvisor/pid/rate");
double bw_adv = pidAdvisor.getDouble("pidAdvisor/bw");

System.Diagnostics.Trace.WriteLine(p_adv, "P");
System.Diagnostics.Trace.WriteLine(i_adv, "I");
System.Diagnostics.Trace.WriteLine(d_adv, "D");
System.Diagnostics.Trace.WriteLine(dlimittimeconstant_adv, "D_tc");
System.Diagnostics.Trace.WriteLine(rate_adv, "rate");
System.Diagnostics.Trace.WriteLine(bw_adv, "bw");

// copy the values from the Advisor to the device
pidAdvisor.setInt("pidAdvisor/todevice", 1);

// Get all calculated parameters.
Lookup result = pidAdvisor.get("pidAdvisor/*");

// extract bode plot and step response
double[] grid = result["/bode"][0].advisorWaves[0].grid;
double[] x = result["/bode"][0].advisorWaves[0].x;
double[] y = result["/bode"][0].advisorWaves[0].y;
String fileName = Environment.CurrentDirectory + "/pidAdvisor.txt";
System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
for (int i = 0; i < grid.Length; ++i)
{
    file.WriteLine("{0} {1} {2}", grid[i], x[i], y[i]);
}
file.Close();
```



```
        Assert.AreEqual(1.0, pidAdvisor.progress());
        Assert.AreNotEqual(0, grid.Length);
    }

    static double Sinc(double x)
    {
        return x != 0.0 ? Math.Sin(Math.PI * x) / (Math.PI * x) : 1.0;
    }

    // ExampleAwgModule shows the usage of the AWG module.
    // It uses the AWG sequencer to generate a wave form.
    // The defined waveform is applied, measured and the
    // results are written to a file.
    [TestMethod, Timeout(10000)]
    public void ExampleAwgModule()
    {
        ziDotNET daq = connect();
        resetDeviceToDefault(daq);

        // check device type, option
        if (!isDeviceFamily(daq, "UHFAWG") && !isDeviceFamily(daq, "UHFQA") && !
hasOption(daq, "AWG"))
        {
            return;
        }

        // Create instrument configuration: disable all outputs, demods and scopes.
        daq.setInt(String.Format("/{0}/demods/*/enable", dev), 0);
        daq.setInt(String.Format("/{0}/demods/*/trigger", dev), 0);
        daq.setInt(String.Format("/{0}/sigouts/*/enables/*", dev), 0);
        daq.setInt(String.Format("/{0}/scopes/*/enable", dev), 0);
        if (hasOption(daq, "IA"))
        {
            daq.setInt(String.Format("/{0}/imps/*/enable", dev), 0);
        }
        daq.sync();

        // Now configure the instrument for this experiment. The following channels
        // and indices work on all device configurations. The values below may be
        // changed if the instrument has multiple input/output channels and/or either
        // the Multifrequency or Multidemodulator options installed.
        int in_channel = 0;
        double frequency = 1e6;
        double amp = 1.0;

        daq.setDouble(String.Format("/{0}/sigouts/0/amplitudes/*", dev), 0.0);
        daq.sync();

        daq.setInt(String.Format("/{0}/sigins/0/imp50", dev), 1);
        daq.setInt(String.Format("/{0}/sigins/0/ac", dev), 0);
        daq.setInt(String.Format("/{0}/sigins/0/diff", dev), 0);
        daq.setInt(String.Format("/{0}/sigins/0/range", dev), 1);
        daq.setDouble(String.Format("/{0}/oscs/0/freq", dev), frequency);
        daq.setInt(String.Format("/{0}/sigouts/0/on", dev), 1);
        daq.setInt(String.Format("/{0}/sigouts/0/range", dev), 1);
        daq.setInt(String.Format("/{0}/sigouts/0/enables/3", dev), 1);
        daq.setDouble(String.Format("/{0}/awgs/0/outputs/0/amplitude", dev), amp);
        daq.setInt(String.Format("/{0}/awgs/0/outputs/0/mode", dev), 0);
        daq.setInt(String.Format("/{0}/awgs/0/time", dev), 0);
        daq.setInt(String.Format("/{0}/awgs/0/userregs/0", dev), 0);

        daq.sync();

        // Number of points in AWG waveform
        int AWG_N = 2000;
```

```

    // Define an AWG program as a string stored in the variable awg_program,
    equivalent to what would
    // be entered in the Sequence Editor window in the graphical UI.
    // This example demonstrates four methods of definig waveforms via the API
    // - (wave w0) loaded directly from programmatically generated CSV file
    wave0.csv.
    //          Waveform shape: Blackman window with negative amplitude.
    // - (wave w1) using the waveform generation functionalities available in the
    AWG Sequencer language.
    //          Waveform shape: Gaussian function with positive amplitude.
    // - (wave w2) using the vect() function and programmatic string replacement.
    //          Waveform shape: Single period of a sine wave.
    string awg_program =
        "const AWG_N = _c1_;\n" +
        "wave w0 = \"wave0\";\n" +
        "wave w1 = gauss(AWG_N, AWG_N/2, AWG_N/20);\n" +
        "wave w2 = vect(_w2_);\n" +
        "wave w3 = zeros(AWG_N);\n" +
        "setTrigger(1);\n" +
        "setTrigger(0);\n" +
        "playWave(w0);\n" +
        "playWave(w1);\n" +
        "playWave(w2);\n" +
        "playWave(w3);\n";

    // Reference waves

    // Define an array of values that are used to write values for wave w0 to a CSV
    file in the
    // module's data directory (Blackman windows)
    var waveform_0 = Enumerable.Range(0, AWG_N).Select(
        v => -1.0 * (0.42 - 0.5 * Math.Cos(2.0*Math.PI * v / (AWG_N-1))) + 0.08 *
        Math.Cos(4*Math.PI*v / (AWG_N - 1))));
    double width = AWG_N / 20;
    var linspace = Enumerable.Range(0, AWG_N).Select(
        v => (v * AWG_N / ((double)AWG_N - 1.0d)) - AWG_N / 2);
    var waveform_1 = linspace.Select(
        v => Math.Exp(-v * v / (2 * width * width)));
    linspace = Enumerable.Range(0, AWG_N).Select(
        v => (v * 2 * Math.PI / ((double)AWG_N - 1.0d)));
    var waveform_2 = linspace.Select(
        v => Math.Sin(v));
    linspace = Enumerable.Range(0, AWG_N).Select(
        v => (v * 12 * Math.PI / ((double)AWG_N - 1.0d)) - 6 * Math.PI);
    var waveform_3 = linspace.Select(
        v => Sinc(v));

    // concatenated reference wave
    double f_s = 1.8e9; // sampling rate of scope and AWG
    double full_scale = 0.75;
    var y_expected =
    waveform_0.Concat(waveform_1).Concat(waveform_2).Concat(waveform_3).Select(
        v => v * full_scale * amp).ToArray();
    var x_expected = Enumerable.Range(0, 4 * AWG_N).Select(v => v / f_s).ToArray();

    // Replace placeholders in program
    awg_program = awg_program.Replace("_w2_", string.Join(",", waveform_2));
    awg_program = awg_program.Replace("_c1_", AWG_N.ToString());

    // Create an instance of the AWG Module
    ziModule awgModule = daq.awgModule();
    awgModule.setByte("awgModule/device", dev);
    awgModule.execute();

    // Get the modules data directory
    string data_dir = awgModule.getString("awgModule/directory");

```

```
// All CSV files within the waves directory are automatically recognized by the
AWG module
data_dir = data_dir + "\\awg\\waves";
if (!Directory.Exists(data_dir)) {
    // The data directory is created by the AWG module and should always exist.
    If this exception is raised,
        // something might be wrong with the file system.
        Assert.Fail(String.Format("AWG module wave directory {0} does not exist or is
not a directory", data_dir));
}
// Save waveform data to CSV
string csv_file = data_dir + "\\wave0.csv";
File.WriteAllText(csv_file, string.Join(",", waveform_0));

// Transfer the AWG sequence program. Compilation starts automatically.
// Note: when using an AWG program from a source file (and only then), the
//       compiler needs to be started explicitly with
//       awgModule.set("awgModule/compiler/start", 1)
awgModule.setByte("awgModule/compiler/sourcestring", awg_program);
while (awgModule.getInt("awgModule/compiler/status") == -1)
{
    System.Threading.Thread.Sleep(100);
}

// check compiler result
long status = awgModule.getInt("awgModule/compiler/status");
if (status == 1)
{
    // compilation failed
    String message = awgModule.getString("awgModule/compiler/statusstring");
    System.Diagnostics.Trace.WriteLine(message, "Compiler message:");
    Assert.Fail("Compilation failed.");
}
if (status == 0)
{
    System.Diagnostics.Trace.WriteLine("Compilation successful with no warnings"
+
    ", will upload the program to the instrument.");
}
if (status == 2)
{
    System.Diagnostics.Trace.WriteLine("Compilation successful with warnings" +
    ", will upload the program to the instrument.");
    String message = awgModule.getString("awgModule/compiler/statusstring");
    System.Diagnostics.Trace.WriteLine("Compiler warning:");
    System.Diagnostics.Trace.WriteLine(message);
}

// wait for waveform upload to finish
while (awgModule.getDouble("awgModule/progress") < 1.0)
{
    System.Diagnostics.Trace.WriteLine(
        awgModule.getDouble("awgModule/progress"), "Progress");
    System.Threading.Thread.Sleep(100);
}

// Replace w3 with waveform_3 using vector write.
// The set command below on awgs/0/waveform/index defines the index of the
waveform in the sequencer program to
// replace with the data that is written to awgs/0/waveform/data.
// Let N be the total number of waveforms and M>0 be the number of waveforms
defined from CSV file. Then the index
// of the waveform to be replaced is defined as following:
// - 0,...,M-1 for all waveforms defined from CSV file alphabetically ordered
by filename,
// - M,...,N-1 in the order that the waveforms are defined in the sequencer
program.
```

```
// For the case of M=0, the index is defined as:
// - 0,...,N-1 in the order that the waveforms are defined in the sequencer
program.
// Of course, for the trivial case of 1 waveform, use index=0 to replace it.
// Here we replace waveform w3, the 4th waveform defined in the sequencer
program. Using 0-based indexing the
// index of the waveform we want to replace (w3, a vector of zeros) is 3:
daq.setInt(String.Format("/{0}/awgs/0/waveform/index", dev), 3);
daq.sync();
// Write the waveform to the memory. For the transferred array, floating-point
(-1.0...+1.0)
// as well as integer (-32768...+32768) data types are accepted.
// For dual-channel waves, interleaving is required.
daq.vectorWrite(String.Format("/{0}/awgs/0/waveform/data", dev),
waveform_3.ToArray<double>());

// Configure the Scope for measurement
daq.setInt(
    String.Format("/{0}/scopes/0/channels/0/inputselect", dev), in_channel);
daq.setInt(String.Format("/{0}/scopes/0/time", dev), 0);
daq.setInt(String.Format("/{0}/scopes/0/enable", dev), 0);
daq.setInt(String.Format("/{0}/scopes/0/length", dev), 16836);

// Now configure the scope's trigger to get aligned data.
daq.setInt(String.Format("/{0}/scopes/0/trigenable", dev), 1);
// Here we trigger on UHF signal input 1. If the instrument has the DIG Option
installed we could
// trigger the scope using an AWG Trigger instead (see the `setTrigger(1);`
line in `awg_program` above).
// 0: Signal Input 1
// 192: AWG Trigger 1
long trigchannel = 0;
daq.setInt(String.Format("/{0}/scopes/0/trigchannel", dev), trigchannel);
if (trigchannel == 0)
{
    // Trigger on the falling edge of the negative blackman waveform `w0` from
our AWG program.
    daq.setInt(String.Format("/{0}/scopes/0/trigslope", dev), 2);
    daq.setDouble(String.Format("/{0}/scopes/0/triglevel", dev), -0.600);

    // Set hysteresis triggering threshold to avoid triggering on noise
    // 'trighysteresis/mode' :
    // 0 - absolute, use an absolute value ('scopes/0/trighysteresis/absolute')
    // 1 - relative, use a relative value ('scopes/0trighysteresis/relative') of
the trigchannel's input range
    // (0.1=10%).
    daq.setDouble(String.Format("/{0}/scopes/0/trighysteresis/mode", dev), 0);
    daq.setDouble(String.Format("/{0}/scopes/0/trighysteresis/relative", dev),
0.025);

    // Set a negative trigdelay to capture the beginning of the waveform.
    daq.setDouble(String.Format("/{0}/scopes/0/trigdelay", dev), -1.0e-6);
}
else
{
    // Assume we're using an AWG Trigger, then the scope configuration is simple:
Trigger on rising edge.
    daq.setInt(String.Format("/{0}/scopes/0/trigslope", dev), 1);

    // Set trigdelay to 0.0: Start recording from when the trigger is activated.
    daq.setDouble(String.Format("/{0}/scopes/0/trigdelay", dev), 0.0);
}

// the trigger reference position relative within the wave, a value of 0.5
corresponds to the center of the wave
daq.setDouble(String.Format("/{0}/scopes/0/trigreference", dev), 0.0);
```

```
// Set the hold off time in-between triggers.
daq.setDouble(String.Format("/{0}/scopes/0/trigholdoff", dev), 0.025);

// Set up the Scope Module.
ziModule scopeModule = daq.scopeModule();
scopeModule.setInt("scopeModule/mode", 1);
scopeModule.subscribe(String.Format("/{0}/scopes/0/wave", dev));
daq.setInt(String.Format("/{0}/scopes/0/single", dev), 1);
scopeModule.execute();
daq.setInt(String.Format("/{0}/scopes/0/enable", dev), 1);
daq.sync();
System.Threading.Thread.Sleep(100);

// Start the AWG in single-shot mode
daq.setInt(String.Format("/{0}/awgs/0/single", dev), 1);
daq.setInt(String.Format("/{0}/awgs/0/enable", dev), 1);

// Read the scope data (manual timeout of 1 second)
double local_timeout = 1.0;
while (scopeModule.progress() < 1.0 && local_timeout > 0.0)
{
    System.Diagnostics.Trace.WriteLine(
        scopeModule.progress() * 100.0, "Scope Progress");
    System.Threading.Thread.Sleep(20);
    local_timeout -= 0.02;
}
string path = String.Format("/{0}/scopes/0/wave", dev);
Lookup lookup = scopeModule.read();
ZIScopeWave[] scopeWaves1 = lookup[path][0].scopeWaves;
float[,] y_measured_in = SimpleValue.getFloatVec2D(scopeWaves1[0].wave);
float[] y_measured = new float[y_measured_in.Length];
for (int i = 0; i < y_measured_in.Length; ++i)
{
    y_measured[i] = y_measured_in[0, i];
}

var x_measured = Enumerable.Range(0, y_measured.Length).Select(
    v => -(long)v * scopeWaves1[0].header.dt +
        (scopeWaves1[0].header.timeStamp -
         scopeWaves1[0].header.triggerTimeStamp) / f_s
    ).ToArray();

// write signals to files
String fileName = Environment.CurrentDirectory + "/awg_measured.txt";
System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
file.WriteLine("t [ns], measured signal [V]");
for (int i = 0; i < y_measured.Length; ++i)
{
    file.WriteLine("{0} {1}", x_measured[i] * 1e9, y_measured[i]);
}
file.Close();

fileName = Environment.CurrentDirectory + "/awg_expected.txt";
file = new System.IO.StreamWriter(fileName);
file.WriteLine("t [ns], expected signal [V]");
for (int i = 0; i < y_expected.Length; ++i)
{
    file.WriteLine("{0} {1}", x_expected[i] * 1e9, y_expected[i]);
}
file.Close();

// checks
Assert.AreNotEqual(0, x_measured.Length);
Assert.AreNotEqual(0, y_measured.Length);

// find minimal difference
double dMinMax = 1e10;
```

```
for (int i = 0; i < x_measured.Length - x_expected.Length; i++)
{
    double dMax = 0;
    for (int k = 0; k < x_expected.Length; k++)
    {
        double d = Math.Abs(y_expected[k] - y_measured[k+i]);
        if (d > dMax)
        {
            dMax = d;
        }
    }

    if (dMax < dMinMax)
    {
        dMinMax = dMax;
    }
}
Assert.IsTrue(dMinMax < 0.1);
}

// ExampleAtutorangingImpedance shows how to perform a manually triggered
// autoranging for impedance while working in manual range mode.
[TestMethod, Timeout(25000)]
public void ExampleAtutorangingImpedance()
{
    ziDotNET daq = connect();
    resetDeviceToDefault(daq);

    // check device type, option
    if (!hasOption(daq, "IA"))
    {
        return;
    }

    // Create instrument configuration: disable all outputs, demods and scopes.
    daq.setInt(String.Format("/{0}/demods/*/enable", dev), 0);
    daq.setInt(String.Format("/{0}/demods/*/trigger", dev), 0);
    daq.setInt(String.Format("/{0}/sigouts/*/enables/*", dev), 0);
    daq.setInt(String.Format("/{0}/scopes/*/enable", dev), 0);
    daq.setInt(String.Format("/{0}/imps/*/enable", dev), 0);
    daq.sync();

    int imp = 0;
    long curr = daq.getInt(String.Format("/{0}/imps/{1}/current/inputselect", dev,
imp));
    long volt = daq.getInt(String.Format("/{0}/imps/{1}/voltage/inputselect", dev,
imp));
    double manCurrRange = 10e-3;
    double manVoltRange = 10e-3;

    // Now configure the instrument for this experiment. The following channels and
    // indices work on all devices with IA option.
    // The values below may be changed if the instrument has multiple IA modules.
    daq.setInt(String.Format("/{0}/imps/{1}/enable", dev, imp), 1);
    daq.setInt(String.Format("/{0}/imps/{1}/mode", dev, imp), 0);
    daq.setInt(String.Format("/{0}/imps/{1}/auto/output", dev, imp), 1);
    daq.setInt(String.Format("/{0}/imps/{1}/auto/bw", dev, imp), 1);
    daq.setDouble(String.Format("/{0}/imps/{1}/freq", dev, imp), 500);
    daq.setInt(String.Format("/{0}/imps/{1}/auto/inputrange", dev, imp), 0);
    daq.setDouble(String.Format("/{0}/currins/{1}/range", dev, curr),
manCurrRange);
    daq.setDouble(String.Format("/{0}/sigins/{1}/range", dev, volt), manVoltRange);
    daq.sync();

    // After setting the device in manual ranging mode we want to trigger manually
    // a one time auto ranging to find a suitable range.
```

```
// Therefore, we trigger the auto ranging for the current input as well as for
the voltage input.
daq.setInt(String.Format("/{0}/currins/{1}/autorange", dev, curr), 1);
daq.setInt(String.Format("/{0}/sigins/{1}/autorange", dev, volt), 1);

// The auto ranging takes some time. We do not want to continue before the best
range is found.
// Therefore, we implement a loop to check if the auto ranging is finished.
int count = 0;
System.Threading.Thread.Sleep(100);
bool finished = false;
var watch = System.Diagnostics.Stopwatch.StartNew();
while (!finished)
{
    ++count;
    System.Threading.Thread.Sleep(500);
    finished = (daq.getInt(String.Format("/{0}/currins/{1}/autorange", dev,
curr)) == 0 &&
                daq.getInt(String.Format("/{0}/sigins/{1}/autorange", dev, volt))
== 0);
}
watch.Stop();
System.Diagnostics.Trace.WriteLine(
    String.Format("Auto ranging finished after {0} s.",
watch.ElapsedMilliseconds / 1e3));

double autoCurrRange = daq.getDouble(String.Format("/{0}/currins/{1}/range",
dev, curr));
double autoVoltRange = daq.getDouble(String.Format("/{0}/sigins/{1}/range",
dev, volt));
System.Diagnostics.Trace.WriteLine(
    String.Format("Current range changed from {0} A to {1} A.", manCurrRange,
autoCurrRange));
System.Diagnostics.Trace.WriteLine(
    String.Format("Voltage range changed from {0} A to {1} A.", manVoltRange,
autoVoltRange));
Assert.IsTrue(count > 1);
}

// ExampleDataAcquisition uses the new data acquisition module to record data
// and writes the result in to a file.
[TestMethod, Timeout(20000)]
public void ExampleDataAcquisition()
{
    ziDotNET daq = connect();
    if ((isDeviceFamily(daq, "MF") && !hasOption(daq, "MD")) ||
(isDeviceFamily(daq, "HF2") && !hasOption(daq, "MFK")))
    {
        daq.disconnect();
        return;
    }

    resetDeviceToDefault(daq);
    daq.setInt(String.Format("/{0}/demods/0/oscsselect", dev), 0);
    daq.setInt(String.Format("/{0}/demods/1/oscsselect", dev), 1);
    daq.setDouble(String.Format("/{0}/oscs/0/freq", dev), 2e6);
    daq.setDouble(String.Format("/{0}/oscs/1/freq", dev), 2.0001e6);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/*", dev), 0);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/0", dev), 1);
    daq.setInt(String.Format("/{0}/sigouts/0/enables/1", dev), 1);
    daq.setInt(String.Format("/{0}/sigouts/0/on", dev), 1);
    daq.setDouble(String.Format("/{0}/sigouts/0/amplitudes/0", dev), 0.2);
    daq.setDouble(String.Format("/{0}/sigouts/0/amplitudes/1", dev), 0.2);
    ziModule trigger = daq.dataAcquisitionModule();
    trigger.setInt("dataAcquisitionModule/grid/mode", 4);
    double demodRate = daq.getDouble(String.Format("/{0}/demods/0/rate", dev));
    double duration = trigger.getDouble("dataAcquisitionModule/duration");
```

```
Int64 sampleCount = System.Convert.ToInt64(demodRate * duration);
trigger.setInt("dataAcquisitionModule/grid/cols", sampleCount);
trigger.setByte("dataAcquisitionModule/device", dev);
trigger.setInt("dataAcquisitionModule/type", 1);
trigger.setDouble("dataAcquisitionModule/level", 0.1);
trigger.setDouble("dataAcquisitionModule/hysteresis", 0.01);
trigger.setDouble("dataAcquisitionModule/bandwidth", 0.0);
String path = String.Format("/{0}/demods/0/sample.r", dev);
trigger.subscribe(path);
String triggerPath = String.Format("/{0}/demods/0/sample.R", dev);
trigger.setByte("dataAcquisitionModule/triggernode", triggerPath);
trigger.execute();
while (!trigger.finished())
{
    System.Threading.Thread.Sleep(100);
    double progress = trigger.progress() * 100;
    System.Diagnostics.Trace.WriteLine(progress, "Progress");
}
Lookup lookup = trigger.read();
ZIDoubleData[] demodSample = lookup[path][0].doubleData;
String fileName = Environment.CurrentDirectory + "/dataacquisition.txt";
System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);
ZIChunkHeader header = lookup[path][0].header;
// Raw system time is the number of microseconds since linux epoch
file.WriteLine("Raw System Time: {0}", header.systemTime);
// Use the utility function ziSystemTimeToDateTime to convert to DateTime
of .NET
    file.WriteLine("Converted System Time: {0}",
ziUtility.ziSystemTimeToDateTime(lookup[path][0].header.systemTime));
    file.WriteLine("Created Timestamp: {0}", header.createdTimeStamp);
    file.WriteLine("Changed Timestamp: {0}", header.changedTimeStamp);
    file.WriteLine("Flags: {0}", header.flags);
    file.WriteLine("Name: {0}", header.name);
    file.WriteLine("Status: {0}", header.status);
    file.WriteLine("Group Index: {0}", header.groupIndex);
    file.WriteLine("Color: {0}", header.color);
    file.WriteLine("Active Row: {0}", header.activeRow);
    file.WriteLine("Trigger Number: {0}", header.triggerNumber);
    file.WriteLine("Grid Rows: {0}", header.gridRows);
    file.WriteLine("Grid Cols: {0}", header.gridCols);
    file.WriteLine("Grid Mode: {0}", header.gridMode);
    file.WriteLine("Grid Operation: {0}", header.gridOperation);
    file.WriteLine("Grid Direction: {0}", header.gridDirection);
    file.WriteLine("Grid Repetitions: {0}", header.gridRepetitions);
    file.WriteLine("Grid Col Delta: {0}", header.gridColDelta);
    file.WriteLine("Grid Col Offset: {0}", header.gridColOffset);
    file.WriteLine("Bandwidth: {0}", header.bandwidth);
    file.WriteLine("Center: {0}", header.center);
    file.WriteLine("NENBW: {0}", header.nenbw);
    for (int i = 0; i < demodSample.Length; ++i)
    {
        file.WriteLine("{0}", demodSample[i].value);
    }
    file.Close();

    Assert.AreEqual(1, trigger.progress());
    Assert.AreNotEqual(0, demodSample.Length);

    daq.disconnect();
}

// ExampleMultiDeviceDataAcquisition
//
// Run the example: Capture demodulator data from two devices using the Data
Acquisition module.
// The devices are first synchronized using the MultiDeviceSync Module.
//
```



```
// Hardware configuration:
// The cabling of the instruments must follow the MDS cabling depicted in
// the MDS tab of LabOne.
// Additionally, Signal Out 1 of the master device is split into Signal In 1 of
the master and slave.
//
[Ignore] // test ignored because it requires special device setup
[TestMethod, Timeout(25000)]
public void ExampleMultiDeviceDataAcquisition()
{
    String[] device_ids = { "dev3133", "dev3144" };

    ziDotNET daq = new ziDotNET();
    daq.init("localhost", 8004, zhinst.ZIAPIVersion_enum.ZI_API_VERSION_6);
    apiServerVersionCheck(daq);
    daq.connectDevice(device_ids[0], "1gbe", "");
    daq.connectDevice(device_ids[1], "1gbe", "");

    // Create instrument configuration: disable all outputs, demods and scopes.
    foreach (String dev in device_ids) {
        daq.setInt(String.Format("/{0}/demods/*/enable", dev), 0);
        daq.setInt(String.Format("/{0}/demods/*/trigger", dev), 0);
        daq.setInt(String.Format("/{0}/sigouts/*/enables/*", dev), 0);
        daq.setInt(String.Format("/{0}/scopes/*/enable", dev), 0);
        daq.setInt(String.Format("/{0}/imps/*/enable", dev), 0);
        daq.sync();
    }

    System.Diagnostics.Trace.WriteLine("Synchronizing devices " + String.Join(",",
device_ids) + "...\\n");

    ziModule mds = daq.multiDeviceSyncModule();
    mds.setInt("multiDeviceSyncModule/start", 0);
    mds.setInt("multiDeviceSyncModule/group", 0);
    mds.execute();
    mds.setString("multiDeviceSyncModule/devices", String.Join(",", device_ids));
    mds.setInt("multiDeviceSyncModule/start", 1);

    // Wait for MDS to complete
    double local_timeout = 20.0;
    long status = 0;
    while (status != 2 && local_timeout > 0.0)
    {
        status = mds.getInt("multiDeviceSyncModule/status");
        System.Threading.Thread.Sleep(100);
        local_timeout -= 0.1;
    }

    if (status != 2)
    {
        System.Diagnostics.Trace.WriteLine("Error during synchronization.\\n");
        Assert.Fail();
    }
    System.Diagnostics.Trace.WriteLine("Devices successfully synchronized.");

    // Device settings
    int demod_c = 0; // demod channel, for paths on the device
    int out_c = 0; // signal output channel
    int out_mixer_c = 0;
    int in_c = 0; // signal input channel
    int osc_c = 0; // oscillator

    double time_constant = 1.0e-3; // [s]
    double demod_rate = 10e3; // [Sa/s]
    int filter_order = 8;
    double osc_freq = 1e3; // [Hz]
    double out_amp = 0.600; // [V]
```

```
// Device settings
foreach (String dev in device_ids)
{
    daq.setDouble(String.Format("/{0}/demods/{1}/phaseshift", dev, demod_c), 0);
    daq.setInt(String.Format("/{0}/demods/{1}/order", dev, demod_c),
filter_order);
    daq.setDouble(String.Format("/{0}/demods/{1}/rate", dev, demod_c),
demod_rate);
    daq.setInt(String.Format("/{0}/demods/{1}/harmonic", dev, demod_c), 1);
    daq.setInt(String.Format("/{0}/demods/{1}/enable", dev, demod_c), 1);
    daq.setInt(String.Format("/{0}/demods/{1}/oscselect", dev, demod_c), osc_c);
    daq.setInt(String.Format("/{0}/demods/{1}/adcselect", dev, demod_c), in_c);
    daq.setDouble(String.Format("/{0}/demods/{1}/timeconstant", dev, demod_c),
time_constant);
    daq.setDouble(String.Format("/{0}/oscs/{1}/freq", dev, osc_c), osc_freq);
    daq.setInt(String.Format("/{0}/sigins/{1}/imp50", dev, in_c), 1);
    daq.setInt(String.Format("/{0}/sigins/{1}/ac", dev, in_c), 0);
    daq.setDouble(String.Format("/{0}/sigins/{1}/range", dev, in_c), out_amp /
2);
}
// settings on master
daq.setInt(String.Format("/{0}/sigouts/{1}/on", device_ids[0], out_c), 1);
daq.setDouble(String.Format("/{0}/sigouts/{1}/range", device_ids[0], out_c),
1);
    daq.setDouble(String.Format("/{0}/sigouts/{1}/amplitudes/{2}", device_ids[0],
out_c, out_mixer_c), out_amp);
    daq.setDouble(String.Format("/{0}/sigouts/{1}/enables/{2}", device_ids[0],
out_c, out_mixer_c), 0);

// Synchronization
daq.sync();

// measuring the transient state of demodulator filters using DAQ module

// DAQ module
// Create a Data Acquisition Module instance, the return argument is a handle
to the module
ziModule daqMod = daq.dataAcquisitionModule();
// Configure the Data Acquisition Module
// Device on which trigger will be performed
daqMod.setString("dataAcquisitionModule/device", device_ids[0]);
// The number of triggers to capture (if not running in endless mode).
daqMod.setInt("dataAcquisitionModule/count", 1);
daqMod.setInt("dataAcquisitionModule/endless", 0);
// 'dataAcquisitionModule/grid/mode' - Specify the interpolation method of
// the returned data samples.
//
// 1 = Nearest. If the interval between samples on the grid does not match
// the interval between samples sent from the device exactly, the nearest
// sample (in time) is taken.
//
// 2 = Linear interpolation. If the interval between samples on the grid does
// not match the interval between samples sent from the device exactly,
// linear interpolation is performed between the two neighbouring
// samples.
//
// 4 = Exact. The subscribed signal with the highest sampling rate (as sent
// from the device) defines the interval between samples on the DAQ
// Module's grid. If multiple signals are subscribed, these are
// interpolated onto the grid (defined by the signal with the highest
// rate, "highest_rate"). In this mode, dataAcquisitionModule/duration is
// read-only and is defined as num_cols/highest_rate.
int grid_mode = 2;
daqMod.setInt("dataAcquisitionModule/grid/mode", grid_mode);
// type:
```

```
//      NO_TRIGGER = 0
//      EDGE_TRIGGER = 1
//      DIGITAL_TRIGGER = 2
//      PULSE_TRIGGER = 3
//      TRACKING_TRIGGER = 4
//      HW_TRIGGER = 6
//      TRACKING_PULSE_TRIGGER = 7
//      EVENT_COUNT_TRIGGER = 8
daqMod.setInt("dataAcquisitionModule/type", 1);
//      triggernode, specify the triggernode to trigger on.
//      SAMPLE.X = Demodulator X value
//      SAMPLE.Y = Demodulator Y value
//      SAMPLE.R = Demodulator Magnitude
//      SAMPLE.THETA = Demodulator Phase
//      SAMPLE.AUXIN0 = Auxilliary input 1 value
//      SAMPLE.AUXIN1 = Auxilliary input 2 value
//      SAMPLE.DIO = Digital I/O value
string triggernode = String.Format("/{0}/demods/{1}/sample.r", device_ids[0],
demod_c);
daqMod.setString("dataAcquisitionModule/triggernode", triggernode);
//      edge:
//      POS_EDGE = 1
//      NEG_EDGE = 2
//      BOTH_EDGE = 3
daqMod.setInt("dataAcquisitionModule/edge", 1);
demod_rate = daq.getDouble(String.Format("/{0}/demods/{1}/rate", device_ids[0],
demod_c));
// Exact mode: To preserve our desired trigger duration, we have to set
// the number of grid columns to exactly match.
double trigger_duration = time_constant * 30;
int sample_count = Convert.ToInt32(demod_rate * trigger_duration);
daqMod.setInt("dataAcquisitionModule/grid/cols", sample_count);
// The length of each trigger to record (in seconds).
daqMod.setDouble("dataAcquisitionModule/duration", trigger_duration);
daqMod.setDouble("dataAcquisitionModule/delay", -trigger_duration / 4);
// Do not return overlapped trigger events.
daqMod.setDouble("dataAcquisitionModule/holdoff/time", 0);
daqMod.setDouble("dataAcquisitionModule/holdoff/count", 0);
daqMod.setDouble("dataAcquisitionModule/level", out_amp / 6);
// The hysteresis is effectively a second criteria (if non-zero) for triggering
// and makes triggering more robust in noisy signals. When the trigger `level`
// is violated, then the signal must return beneath (for positive trigger edge)
// the hysteresis value in order to trigger.
daqMod.setDouble("dataAcquisitionModule/hysteresis", 0.01);
// synchronizing the settings
daq.sync();

// Recording

// Subscribe to the demodulators
daqMod.unsubscribe("");
foreach (String dev in device_ids)
{
    string node = String.Format("/{0}/demods/{1}/sample.r", dev, demod_c);
    daqMod.subscribe(node);
}

// Execute the module
daqMod.execute();
// Send a trigger
daq.setDouble(String.Format("/{0}/sigouts/{1}/enables/{2}", device_ids[0],
out_c, out_mixer_c), 1);
while (!daqMod.finished())
{
    System.Threading.Thread.Sleep(1000);
    System.Diagnostics.Trace.WriteLine(String.Format("Progress {0}",
daqMod.progress()));
}
```

```
    }

    // Read the result
    Lookup result = daqMod.read();

    // Turn off the trigger
    daq.setDouble(String.Format("/{0}/sigouts/{1}/enables/{2}", device_ids[0],
out_c, out_mixer_c), 0);
    // Finish the DAQ module
    daqMod.finish();
    daqMod.clear();

    // Stopping the MDS module
    mds.clear();

    // Extracting and saving the data
    double mClockbase = daq.getDouble(String.Format("/{0}/clockbase",
device_ids[0]));

    List<ZIDoubleData[]> data = new List<ZIDoubleData[]>();
    foreach (String dev in device_ids)
    {
        string node = string.Format("/{0}/demods/{1}/sample.r", dev, demod_c);
        data.Add(result[node][0].doubleData);
    }

    String fileName = Environment.CurrentDirectory + "/mds_dataacquisition.txt";
    System.IO.StreamWriter file = new System.IO.StreamWriter(fileName);

    for (int i = 0; i < data[0].Length; ++i)
    {
        file.WriteLine("{0},{1},{2}", (data[0][i].timeStamp - data[0][0].timeStamp) /
mClockbase,
        data[0][i].value, data[1][i].value);
    }
    file.Close();

    daq.disconnect();
}

}
```

Chapter 8. C Programming

The LabOne C API, also known as ziAPI, provides a simple and robust way to communicate with the Data Server. It enables you to get or set parameters and receive streaming data.

8.1. Getting Started

After installing the LabOne software package and relevant drivers for your instrument you are ready start programming with ziAPI. All you need is a C compiler, linker and editor.

The structure of a program using ziAPI can be split into three parts: initialization/connection, data manipulation and disconnection/cleanup. The basic object that is always used is the ziConnection data structure. First, ziConnection is has to be initialized by calling [ziAPIInit](#). After initialization ziConnection is ready to connect to a ziServer by calling [ziAPIConnect](#). Then ziConnection is ready to be used for getting and setting parameters and streaming data. When ziConnection is not needed anymore the established connection to the ziServer has to be hung up using [ziAPIDisconnect](#) before cleaning it up by calling [ziAPIDestroy](#).

8.1.1. Examples

Along with the LabOne C API DLL, a LabOne installation includes examples to help getting started with the LabOne C API. On Windows they are located in the folder:

```
C:\Program Files\Zurich Instruments\LabOne\API\C\examples\
```

and on Linux, after extracting the LabOne tarball, they are located in the folder:

```
API/C/examples.
```

Below you find a simple program, which sets the demodulator rate of all demods for all devices.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

int main() {
    ZIResult_enum retVal;
    ZIConnection conn;
    char* errBuffer;
    const char serverAddress[] = "localhost";

    // Initialize ZIConnection.
    retVal = ziAPIInit(&conn);
    if (retVal != ZI_INFO_SUCCESS) {
        ziAPIGetError(retVal, &errBuffer, NULL);
        fprintf(stderr, "Can't init Connection: %s\n", errBuffer);
        return 1;
    }

    // Connect to the Data Server: Use port 8005 for the HF2 Data Server, use
    // 8004 for the UHF and MF Data Servers. HF2 only support ZI_API_VERSION_1,
    // see the LabOne Programming Manual for an explanation of API Levels.
    retVal = ziAPIConnectEx(conn, serverAddress, 8004, ZI_API_VERSION_6, NULL);
    if (retVal != ZI_INFO_SUCCESS) {
        ziAPIGetError(retVal, &errBuffer, NULL);
        fprintf(stderr, "Error, can't connect to the Data Server: `%s`.\n", errBuffer);
    } else {
        // Set all demodulator rates of device dev1046 to 150 Hz
        retVal = ziAPISetValueD(conn, "/dev1046/demods/*/rate", 150);
        if (retVal != ZI_INFO_SUCCESS) {
            ziAPIGetError(retVal, &errBuffer, NULL);
            fprintf(stderr, "Can't set parameter: %s\n", errBuffer);
        }

        // Disconnect from the Data Server. Since ZIAPIDisconnect always returns
        // ZI_INFO_SUCCESS no error handling is required.
    }
}
```

```
    ziAPIDisconnect(conn);  
}  
  
// Destroy the ZIConnection. Since ZIAPIDestroy always returns  
// ZI_INFO_SUCCESS, no error handling is required.  
ziAPIDestroy(conn);  
  
return 0;  
}
```

8.2. Module Documentation

8.2.1. Connecting to Data Server

This section describes how to initialize the `ZIConnection` and establish a connection to Data Server as well as how to disconnect after all data handling is done and cleanup the `ZIConnection`.

Typedefs

- `typedef ZIConnection`
The `ZIConnection` is a connection reference; it holds information and helper variables about a connection to the Data Server. There is nothing in this reference which the user may use, so it is hidden and instead a dummy pointer is used. See [ziAPIInit](#) for how to create a `ZIConnection`.

Enumerations

- `enum ZIAPIVersion_enum { ZI_API_VERSION_0, ZI_API_VERSION_1, ZI_API_VERSION_4, ZI_API_VERSION_5, ZI_API_VERSION_6, ZI_API_VERSION_MAX }`

Functions

- `ZIResult_enum ziAPIInit ( ZIConnection* conn )`
Initializes a `ZIConnection` structure.
- `ZIResult_enum ziAPIDestroy ( ZIConnection conn )`
Destroys a `ZIConnection` structure.
- `ZIResult_enum ziAPIConnect ( ZIConnection conn, const char* hostname, uint16_t port )`
Connects the `ZIConnection` to Data Server.
- `ZIResult_enum ziAPIDisconnect ( ZIConnection conn )`
Disconnects an established connection.
- `ZIResult_enum ziAPIListImplementations ( char* implementations, uint32_t bufferSize )`
Returns the list of supported implementations.
- `ZIResult_enum ziAPIConnectEx ( ZIConnection conn, const char* hostname, uint16_t port, ZIAPIVersion_enum apiLevel, const char* implementation )`
Connects to Data Server and enables extended `ziAPI`.
- `ZIResult_enum ziAPIGetConnectionAPILevel ( ZIConnection conn, ZIAPIVersion_enum* apiLevel )`
Returns `ziAPI` level used for the connection `conn`.
- `ZIResult_enum ziAPIGetVersion ( const char** version )`
Retrieves the release version of `ziAPI`.

- `ZIResult_enum` `ziAPIGetRevision` (unsigned int* revision)
Retrieves the revision of ziAPI.

Detailed Description

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

int connection() {
    ZIResult_enum retVal;
    ZIConnection conn;
    char* errBuffer;
    const char serverAddress[] = "localhost";

    // Initialize ZIConnection.
    retVal = ziAPIInit(&conn);
    if (retVal != ZI_INFO_SUCCESS) {
        ziAPIGetError(retVal, &errBuffer, NULL);
        fprintf(stderr, "Can't init Connection: %s\n", errBuffer);
        return 1;
    }

    // Connect to the Data Server: Use port 8005 for the HF2 Data Server, use
    // 8004 for the UHF and MF Data Servers. HF2 only support ZI_API_VERSION_1,
    // see the LabOne Programming Manual for an explanation of API Levels.
    retVal = ziAPIConnectEx(conn, serverAddress, 8004, ZI_API_VERSION_6, NULL);
    if (retVal != ZI_INFO_SUCCESS) {
        ziAPIGetError(retVal, &errBuffer, NULL);
        fprintf(stderr, "Error, can't connect to the Data Server: `%s`.\n", errBuffer);
    } else {
        /*
         * Do something using ZIConnection here.
         */

        // Since ZIAPIDisconnect always returns ZI_INFO_SUCCESS
        // no error handling is required.
        ziAPIDisconnect(conn);
    }

    // Since ZIAPIDestroy always returns ZI_INFO_SUCCESS
    // no error handling is required.
    ziAPIDestroy(conn);

    return 0;
}
```

Enumeration Type Documentation

Enumerator:

- ZI_API_VERSION_0
- ZI_API_VERSION_1
- ZI_API_VERSION_4
- ZI_API_VERSION_5
- ZI_API_VERSION_6
- ZI_API_VERSION_MAX

Function Documentation

ziAPIInit

ZIResult_enum ziAPIInit (**ZIConnection*** conn)

Initializes a **ZIConnection** structure.

This function initializes the structure so that it is ready to connect to Data Server. It allocates memory and sets up the infrastructure needed.

Parameters:

[out] conn
Pointer to **ZIConnection** that is to be initialized

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_MALLOC on memory allocation failure
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDestroy](#), [ziAPIConnect](#), [ziAPIDisconnect](#)

See [Connection](#) for an example

ziAPIDestroy

ZIResult_enum ziAPIDestroy (ZIConnection conn)

Destroys a [ZIConnection](#) structure.

This function frees all memory that has been allocated by [ziAPIInit](#). If it is called with an uninitialized [ZIConnection](#) struct it may result in segmentation faults as well when it is called with a struct for which [ZIAPIDestroy](#) already has been called.

Parameters:

[in] conn

Pointer to [ZIConnection](#) struct that has to be destroyed

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIInit](#), [ziAPIConnect](#), [ziAPIDisconnect](#)

See [Connection](#) for an example

ziAPIConnect

ZIResult_enum ziAPIConnect ([ZIConnection](#) conn, const char* hostname, uint16_t port)

Connects the ZIConnection to Data Server.

Connects to Data Server using a [ZIConnection](#) and prepares for data exchange. For most cases it is enough to just give a reference to the connection and give NULL for hostname and 0 for the port, so it connects to localhost on the default port.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the connection should be established

[in] hostname

Name of the Host to which it should be connected, if NULL "localhost" will be used as default

[in] port

The Number of the port to connect to. If 0, default port of the local Data Server will be used (8005)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_HOSTNAME if the given host name could not be found
- ZI_ERROR_SOCKET_CONNECT if no connection could be established
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_SOCKET_INIT if initialization of the socket failed
- ZI_ERROR_CONNECTION when the Data Server didn't return the correct answer
- ZI_ERROR_TIMEOUT when initial communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDisconnect](#), [ziAPIInit](#), [ziAPIDestroy](#)

See [Connection](#) for an example

ziAPIDisconnect

ZIResult_enum `ziAPIDisconnect ( ZIConnection conn )`

Disconnects an established connection.

Disconnects from Data Server. If the connection has not been established and the function is called it returns without doing anything.

Parameters:

[in] conn
Pointer to ZIConnection to be disconnected

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnect](#), [ziAPIInit](#), [ziAPIDestroy](#)

See [Connection](#) for an example

ziAPIListImplementations

ZIResult_enum **ziAPIListImplementations** (char* implementations, uint32_t bufferSize)

Returns the list of supported implementations.

Returned names are defined by implementations in the linked library and may change depending on software version.

Parameters:

[out] implementations

Pointer to a buffer receiving a newline-delimited list of the names of all the supported ziAPI implementations. The string is zero-terminated.

[in] bufferSize

The size of the buffer assigned to the implementations parameter

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_LENGTH if the length of the char-buffer given by MaxLen is too small for all elements
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#)

ziAPIConnectEx

ZIResult_enum ziAPIConnectEx (**ZIConnection** conn, const char* hostname, uint16_t port, ZIAPIVersion_enum apiLevel, const char* implementation)

Connects to Data Server and enables extended ziAPI.

With apiLevel=ZI_API_VERSION_1 and implementation=NULL, this call is equivalent to plain [ziAPIConnect](#). With other version and implementation values enables corresponding ziAPI extension and connection using different implementation.

Parameters:

[in] conn

Pointer to the ZIConnection with which the connection should be established

[in] hostname

Name of the host to which it should be connected, if NULL "localhost" will be used as default

[in] port

The number of the port to connect to. If 0 the port of the local Data Server will be used

[in] apiLevel

Specifies the ziAPI compatibility level to use for this connection (1 or 4).

[in] implementation

Specifies implementation to use for a connection, must be one of the returned by ziAPIListImplementations or NULL to select default implementation

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_HOSTNAME if the given host name could not be found
- ZI_ERROR_SOCKET_CONNECT if no connection could be established
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_SOCKET_INIT if initialization of the socket failed
- ZI_ERROR_CONNECTION when the Data Server didn't return the correct answer or requested implementation is not found or doesn't support requested ziAPI level
- ZI_ERROR_TIMEOUT when initial communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIListImplementations](#), [ziAPIConnect](#), [ziAPIDisconnect](#), [ziAPIInit](#), [ziAPIDestroy](#), [ziAPIGetConnectionVersion](#)

See [Connection](#) for an example

ziAPIGetConnectionAPILevel

ZIResult_enum ziAPIGetConnectionAPILevel (**ZIConnection** conn, ZIAPIVersion_enum* apiLevel)

Returns ziAPI level used for the connection conn.

Parameters:

[in] conn

Pointer to ZIConnection

[out] apiLevel

Pointer to preallocated ZIAPIVersion_enum, receiving the ziAPI level

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION if level can not be determined due to conn is not connected

See Also:

[ziAPIConnectEx](#), [ziAPIGetVersion](#), [ziAPIGetRevision](#)

ziAPIGetVersion

ZIResult_enum ziAPIGetVersion (const char** version)

Retrieves the release version of ziAPI.

Sets the passed pointer to point to the null-terminated release version string of ziAPI.

Parameters:

[in] version
Pointer to const char pointer.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#), [ziAPIGetRevision](#), [ziAPIGetConnectionAPILevel](#)

ziAPIGetRevision

ZIResult_enum ziAPIGetRevision (unsigned int* revision)

Retrieves the revision of ziAPI.

Sets an unsigned int with the revision (build number) of the ziAPI you are using.

Parameters:

[in] revision

Pointer to an unsigned int to fill up with the revision.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#), [ziAPIGetVersion](#), [ziAPIGetConnectionAPILevel](#)

8.2.2. Tree

All parameters and streams are organized in a tree. You can list the whole tree, parts of it or single items using [ziAPIListNodes](#) or you may update the tree with nodes of newly connected devices by using [ziAPIUpdateDevices](#).

Enumerations

- `enum ZIListNodes_enum { ZI_LIST_NODES_NONE, ZI_LIST_NODES_RECURSIVE, ZI_LIST_NODES_ABSOLUTE, ZI_LIST_NODES_LEAFONLY, ZI_LIST_NODES_SETTINGSONLY, ZI_LIST_NODES_STREAMINGONLY, ZI_LIST_NODES_SUBSCRIBEDONLY, ZI_LIST_NODES_BASECHANNEL, ZI_LIST_NODES_GETONLY }`

Defines the values of the flags used in [ziAPIListNodes](#).

Functions

- `ZIResult_enum` [ziAPIListNodes](#) ([ZIConnection](#) conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)
Returns all child nodes found at the specified path.
- `ZIResult_enum` [ziAPIListNodesJSON](#) ([ZIConnection](#) conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)
Returns all child nodes found at the specified path.

Detailed Description

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void PrintChildren(ZIConnection Conn,
                  char* Path) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    char NodesBuffer[8192];

    if ((RetVal = ziAPIListNodes(Conn,
                                Path,
                                NodesBuffer,
                                8192,
                                ZI\_LIST\_NODES\_NONE)) != ZI\_INFO\_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Can't List Nodes: %s\n", ErrBuffer);
    } else {
        char* Ptr = NodesBuffer;
        char* LastPtr = Ptr;

        // print out each node on a separate line with dash as prefix
        for (; *Ptr != 0; Ptr++) {
            if (*Ptr == '\n') {
                *Ptr = 0;
                printf("- %s\n", LastPtr);
            }
        }
    }
}
```

```
        LastPtr = Ptr + 1;
    }
}

// print out the last node
if (Ptr != LastPtr) {
    printf("- %s\n", LastPtr);
}
}
}
```

Enumeration Type Documentation

Defines the values of the flags used in [ziAPIListNodes](#).

Enumerator:

- `ZI_LIST_NODES_NONE`
Default, return a simple listing of the given node immediate descendants.
- `ZI_LIST_NODES_RECURSIVE`
List the nodes recursively.
- `ZI_LIST_NODES_ABSOLUTE`
Return absolute paths.
- `ZI_LIST_NODES_LEAFSONLY`
Return only leaf nodes, which means the nodes at the outermost level of the tree.
- `ZI_LIST_NODES_SETTINGSONLY`
Return only nodes which are marked as setting.
- `ZI_LIST_NODES_STREAMINGONLY`
Return only streaming nodes (nodes that can be pushed from the device at a high data rate)
- `ZI_LIST_NODES_SUBSCRIBEDONLY`
Return only nodes that are subscribed to in the API session.
- `ZI_LIST_NODES_BASECHANNEL`
- `ZI_LIST_NODES_GETONLY`

Function Documentation

ziAPIListNodes

ZIResult_enum ziAPIListNodes (**ZIConnection** conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child nodes found at the specified path.

This function returns a list of node names found at the specified path. The path may contain wildcards so that the returned nodes do not necessarily have to have the same parents. The list is returned in a null-terminated char-buffer, each element delimited by a newline. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be ZIResult_enum::ZI_LENGTH.

Parameters:

[in] conn

Pointer to the ZIConnection for which the node names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with newline-delimited list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer used for the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in [ZIListNodes_enum](#).

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds [MAX_PATH_LEN](#) or the length of the char-buffer for the nodes given by bufferSize is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Tree Listing](#) for an example

See Also:

ziAPIUpdate

ziAPIListNodesJSON

ZIResult_enum ziAPIListNodesJSON (**ZIConnection** conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child nodes found at the specified path.

This function returns a list of node names found at the specified path, formatted as JSON. The path may contain wildcards so that the returned nodes do not necessarily have to have the same parents. The list is returned in a null-terminated char-buffer. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be ZIResult_enum::ZI_LENGTH.

Parameters:

[in] conn

Pointer to the ZIConnection for which the node names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with JSON-formatted list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer used for the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in [ZIListNodes_enum](#).

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds [MAX_PATH_LEN](#) or the length of the char-buffer for the nodes given by bufferSize is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Tree Listing](#) for an example

See Also:

[ziAPIUpdate](#)

8.2.3. Set and Get Parameters

This section describes several functions for getting and setting parameters of different datatypes.

Functions

- `ZIResult_enum` `ziAPIGetValueD ( ZIConnection conn, const char* path, ZIDoubleData* value )`
gets the double-type value of the specified node
- `ZIResult_enum` `ziAPIGetComplexData ( ZIConnection conn, const char* path, ZIDoubleData* real, ZIDoubleData* imag )`
gets the complex double-type value of the specified node
- `ZIResult_enum` `ziAPIGetValueI ( ZIConnection conn, const char* path, ZIIntegerData* value )`
gets the integer-type value of the specified node
- `ZIResult_enum` `ziAPIGetDemodSample ( ZIConnection conn, const char* path, ZIDemodSample* value )`
Gets the demodulator sample value of the specified node.
- `ZIResult_enum` `ziAPIGetDIOSample ( ZIConnection conn, const char* path, ZIDIOSample* value )`
Gets the Digital I/O sample of the specified node.
- `ZIResult_enum` `ziAPIGetAuxInSample ( ZIConnection conn, const char* path, ZIAuxInSample* value )`
gets the AuxIn sample of the specified node
- `ZIResult_enum` `ziAPIGetValueB ( ZIConnection conn, const char* path, unsigned char* buffer, unsigned int* length, unsigned int bufferSize )`
gets the Bytearray value of the specified node
- `ZIResult_enum` `ziAPIGetValueString ( ZIConnection conn, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize )`
gets a null-terminated string value of the specified node
- `ZIResult_enum` `ziAPIGetValueStringUnicode ( ZIConnection conn, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize )`
gets a null-terminated string value of the specified node
- `ZIResult_enum` `ziAPISetValueD ( ZIConnection conn, const char* path, ZIDoubleData value )`
asynchronously sets a double-type value to one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetComplexData ( ZIConnection conn, const char* path, ZIDoubleData real, ZIDoubleData imag )`
asynchronously sets a double-type complex value to one or more nodes specified in the path

- `ZIResult_enum` `ziAPISetValueI ( ZIConnection conn, const char* path, ZIntegerData value )`
asynchronously sets an integer-type value to one or more nodes specified in a path
- `ZIResult_enum` `ziAPISetValueB ( ZIConnection conn, const char* path, unsigned char* buffer, unsigned int length )`
asynchronously sets the binary-type value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetValueString ( ZIConnection conn, const char* path, const char* str )`
asynchronously sets a string value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetValueStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
asynchronously sets a unicode encoded string value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISyncSetValueD ( ZIConnection conn, const char* path, ZDoubleData* value )`
synchronously sets a double-type value to one or more nodes specified in the path
- `ZIResult_enum` `ziAPISyncSetValueI ( ZIConnection conn, const char* path, ZIntegerData* value )`
synchronously sets an integer-type value to one or more nodes specified in a path
- `ZIResult_enum` `ziAPISyncSetValueB ( ZIConnection conn, const char* path, uint8_t* buffer, uint32_t* length, uint32_t bufferSize )`
Synchronously sets the binary-type value of one ore more nodes specified in the path.
- `ZIResult_enum` `ziAPISyncSetValueString ( ZIConnection conn, const char* path, const char* str )`
Synchronously sets a string value of one or more nodes specified in the path.
- `ZIResult_enum` `ziAPISyncSetValueStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
Synchronously sets a unicode string value of one or more nodes specified in the path.
- `ZIResult_enum` `ziAPISync ( ZIConnection conn )`
Synchronizes the session by dropping all pending data.
- `ZIResult_enum` `ziAPIEchoDevice ( ZIConnection conn, const char* deviceSerial )`
Sends an echo command to a device and blocks until answer is received.
- `__inline ZIResult_enum` `ziAPIGetValueS ( ZIConnection conn, char* path, DemodSample* value )`

- `__inline ZIResult_enum ziAPIGetValueDIO ( ZIConnection conn, char* path, DIOSample* value )`
- `__inline ZIResult_enum ziAPIGetValueAuxIn ( ZIConnection conn, char* path, AuxInSample* value )`

Function Documentation

ziAPIGetValueD

ZIResult_enum ziAPIGetValueD (**ZIConnection** conn, const char* path, ZIDoubleData* value)

gets the double-type value of the specified node

This function retrieves the numerical value of the specified node as an double-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a double in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2018] Zurich Instruments AG
```

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
```

```
#include "ziAPI.h"
#include "ziUtils.h"
```

```
void setDemodRate(ZIConnection conn, const char** deviceId, uint32_t index,
ZIDoubleData rate) {
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPISetValueD(conn, nodePath, rate));
}
```

```
ZIDoubleData getDemodRate(ZIConnection conn, const char** deviceId, uint32_t index) {
    ZIDoubleData rate;
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPIGetValueD(conn, nodePath, &rate));
    return rate;
}

int main() {
    // The device address of the device to run the example on.
    char deviceAddress[] = "dev2006";
    // The maximum API Level supported by this example.
    // Please use ZI_API_VERSION_1 if using an HF2 Instrument.
    ZIAPIVersion_enum apiLevel = ZI_API_VERSION_6;

    // The ZIConnection is actually a pointer.
    ZIConnection conn;
    if (isError(ziAPIInit(&conn))) {
        return 1;
    }

    ziAPISetDebugLevel(0);
    ziAPIWriteDebugLog(0, "Logging enabled.");

    const char *deviceId;
    if (!ziCreateAPISession(conn, deviceAddress, apiLevel, &deviceId)) {
        try {
            ziApiServerVersionCheck(conn);

            // Set a device configuration.
            uint32_t index = 0;
            ZIDoubleData rate = 10e3;
            setDemodRate(conn, &deviceId, index, rate);

            // Read it back.
            rate = getDemodRate(conn, &deviceId, index);
            std::cout << "[INFO] " << "Device " << deviceId << " demod " << index << " has
rate " << rate << ".\n";

        } catch (std::runtime_error& e) {
            char extErrorMessage[1024] = "";
            ziAPIGetLastError(conn, extErrorMessage, 1024);
            fprintf(stderr, "[ERROR] %s\ndetails: `%s`\n.", e.what(), extErrorMessage);
        } catch (...) {
            // Ensure all exceptions are caught.
            fprintf(stderr, "[ERROR] Unexpected error\n.");
        }
        ziAPIDisconnect(conn);
    } else {
        char extErrorMessage[1024] = "";
        ziAPIGetLastError(conn, extErrorMessage, 1024);
        fprintf(stderr, "[ERROR] Details: `%s`\n.", extErrorMessage);
    }
    ziAPIDestroy(conn);

    return 0;
}
```

See Also:

[ziAPISetValueD](#), [ziAPIGetValueAsPollData](#)

ziAPIGetComplexData

ZIResult_enum ziAPIGetComplexData (**ZIConnection** conn, const char* path, **ZIDoubleData*** real, **ZIDoubleData*** imag)

gets the complex double-type value of the specified node

This function retrieves the numerical value of the specified node as an complex double-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] real

Pointer to a double in which the real value should be written

[out] imag

Pointer to a double in which the imaginary value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2018] Zurich Instruments AG
```

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
```

```
#include "ziAPI.h"
#include "ziUtils.h"
```

```
void setDemodRate(ZIConnection conn, const char** deviceId, uint32_t index,
ZIDoubleData rate) {
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "/%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPISetValueD(conn, nodePath, rate));
}
```

```
}

ZIDoubleData getDemodRate(ZIConnection conn, const char** deviceId, uint32_t index) {
    ZIDoubleData rate;
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "/%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPIGetValueD(conn, nodePath, &rate));
    return rate;
}

int main() {
    // The device address of the device to run the example on.
    char deviceAddress[] = "dev2006";
    // The maximum API Level supported by this example.
    // Please use ZI_API_VERSION_1 if using an HF2 Instrument.
    ZIAPIVersion_enum apiLevel = ZI_API_VERSION_6;

    // The ZIConnection is actually a pointer.
    ZIConnection conn;
    if (isError(ziAPIInit(&conn))) {
        return 1;
    }

    ziAPISetDebugLevel(0);
    ziAPIWriteDebugLog(0, "Logging enabled.");

    const char *deviceId;
    if (!ziCreateAPISession(conn, deviceAddress, apiLevel, &deviceId)) {
        try {
            ziApiServerVersionCheck(conn);

            // Set a device configuration.
            uint32_t index = 0;
            ZIDoubleData rate = 10e3;
            setDemodRate(conn, &deviceId, index, rate);

            // Read it back.
            rate = getDemodRate(conn, &deviceId, index);
            std::cout << "[INFO] " << "Device " << deviceId << " demod " << index << " has
rate " << rate << ".\n";

        } catch (std::runtime_error& e) {
            char extErrorMessage[1024] = "";
            ziAPIGetLastError(conn, extErrorMessage, 1024);
            fprintf(stderr, "[ERROR] %s\ndetails: `%s`\n.", e.what(), extErrorMessage);
        } catch (...) {
            // Ensure all exceptions are caught.
            fprintf(stderr, "[ERROR] Unexpected error\n.");
        }
        ziAPIDisconnect(conn);
    } else {
        char extErrorMessage[1024] = "";
        ziAPIGetLastError(conn, extErrorMessage, 1024);
        fprintf(stderr, "[ERROR] Details: `%s`\n", extErrorMessage);
    }
    ziAPIDestroy(conn);

    return 0;
}
```

See Also:

[ziAPISetComplexData](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueI

ZIResult_enum ziAPIGetValueI (**ZIConnection** conn, const char* path, ZIIntegerData* value)

gets the integer-type value of the specified node

This function retrieves the numerical value of the specified node as an integer-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to an 64bit integer in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueI](#), [ziAPIGetValueAsPollData](#)

ziAPIGetDemodSample

ZIResult_enum ziAPIGetDemodSample (**ZIConnection** conn, const char* path, **ZIDemodSample*** value)

Gets the demodulator sample value of the specified node.

This function retrieves the value of the specified node as an **DemodSample** struct. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to paths matching DEMODS/[0-9]+/SAMPLE.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a **ZIDemodSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetSampleS(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIDemodSample DemodSample;

    if ((RetVal = ziAPIGetDemodSample(Conn,
                                     "/dev1046/demods/0/sample",
                                     &DemodSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
```

```
        printf("TS = %f, X=%f, Y=%f\n",
               (float)DemodSample.timeStamp,
               DemodSample.x,
               DemodSample.y);
    }
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetDIOSample

ZIResult_enum ziAPIGetDIOSample (**ZIConnection** conn, const char* path, **ZIDIOSample*** value)

Gets the Digital I/O sample of the specified node.

This function retrieves the newest available DIO sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to nodes ending in "/DIOS/[0-9]+/INPUT".

Parameters:

[in] conn

Pointer to the ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a **ZIDIOSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetDIOSample(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIDIOSample DIOSample;

    if ((RetVal = ziAPIGetDIOSample(Conn,
                                    "/dev1046/dios/0/output",
                                    &DIOSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
```

```
        printf("TS = %f, bits=%08x\n",  
              (float)DIOSample.timeStamp,  
              DIOSample.bits);  
    }  
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetAuxInSample

ZIResult_enum ziAPIGetAuxInSample (**ZIConnection** conn, const char* path, **ZIAuxInSample*** value)

gets the AuxIn sample of the specified node

This function retrieves the newest available AuxIn sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to nodes ending in "/AUXINS/[0-9]+/SAMPLE".

Parameters:

[in] conn

Pointer to the ziConnection with which the Value should be retrieved

[in] path

Path to the Node holding the value

[out] value

Pointer to an **ZIAuxInSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetSampleAuxIn(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIAuxInSample AuxInSample;

    if ((RetVal = ziAPIGetAuxInSample(Conn,
                                     "/dev1046/auxins/0/sample",
                                     &AuxInSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s\n", ErrBuffer);
    }
}
```

```
    } else {  
        printf("TS = %f, ch0=%f, ch1=%f\n",  
              (float)AuxInSample.timeStamp,  
              AuxInSample.ch0,  
              AuxInSample.ch1);  
    }  
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetValueB

ZIResult_enum ziAPIGetValueB (**ZIConnection** conn, const char* path, unsigned char* buffer, unsigned int* length, unsigned int bufferSize)

gets the Bytearray value of the specified node

This function retrieves the newest available DIO sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved data in

[out] length

Pointer to an unsigned int to store the length of data in. if an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void PrintVersion(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;
```



```
const char* Path = "ZI/ABOUT/VERSION";
unsigned char Buffer[0xff];
unsigned int Length;

if ((RetVal = ziAPIGetValueB(Conn,
                             Path,
                             Buffer,
                             &Length,
                             sizeof(Buffer) - 1)) != ZI\_INFO\_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't get value: %s.\n", ErrBuffer);
} else {
    Buffer[Length] = 0;
    printf("%s=\"%s\"\n", Path, Buffer);
}
}
```

See Also:

[ziAPISetValueB](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueString

ZIResult_enum ziAPIGetValueString (**ZIConnection** conn, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize)

gets a null-terminated string value of the specified node

This function retrieves the newest string value for the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueString](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueStringUnicode

ZIResult_enum ziAPIGetValueStringUnicode (**ZIConnection** conn, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize)

gets a null-terminated string value of the specified node

This function retrieves the newest unicode string value for the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] wbuffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueStringUnicode](#), [ziAPIGetValueAsPollData](#)

ziAPISetValueD

ZIResult_enum ziAPISetValueD (**ZIConnection** conn, const char* path, ZIDoubleData value)

asynchronously sets a double-type value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set.

[in] path

Path to the Node(s) for which the value(s) will be set to Value.

[in] value

The double-type value that will be written to the node(s).

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueD](#). [ziAPISyncSetValueD](#)

ziAPISetComplexData

ZIResult_enum ziAPISetComplexData (**ZIConnection** conn, const char* path, ZIDoubleData real, ZIDoubleData imag)

asynchronously sets a double-type complex value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to the complex value (real, imag). More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set. If the node does not support complex values only the real value will be updated.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set.

[in] path

Path to the Node(s) for which the value(s) will be set to Value.

[in] real

The real value that will be written to the node(s).

[in] imag

The imag value that will be written to the node(s).

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetComplexData](#). [ziAPISyncSetComplexData](#)

ziAPISetValueI

ZIResult_enum ziAPISetValueI (**ZIConnection** conn, const char* path, ZIIntegerData value)

asynchronously sets an integer-type value to one or more nodes specified in a path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] value

The int-type value that will be written to the node(s)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueI](#). [ziAPISyncSetValueI](#)

ziAPISetValueB

ZIResult_enum ziAPISetValueB (**ZIConnection** conn, const char* path, unsigned char* buffer, unsigned int length)

asynchronously sets the binary-type value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] buffer

Pointer to the byte array with the data

[in] length

Length of the data in the buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void ProgramCPU(ZIConnection Conn,
                unsigned char* Buffer,
                int Len) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    if ((RetVal = ziAPISetValueB(Conn,
```

```
        "/dev1046/cpus/0/program",
        Buffer,
        Len)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
}
}
```

See Also:

[ziAPIGetValueB](#). [ziAPISyncSetValueB](#)

ziAPISetValueString

ZIResult_enum ziAPISetValueString (**ZIConnection** conn, const char* path, const char* str)

asynchronously sets a string value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] str

Pointer to a null terminated string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueString](#). [ziAPISyncSetValueString](#)

ziAPISetValueStringUnicode

ZIResult_enum ziAPISetValueStringUnicode (**ZIConnection** conn, const char* path, const wchar_t* wstr)

asynchronously sets a unicode encoded string value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] wstr

Pointer to a null terminated unicode string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueStringUnicode](#). [ziAPISyncSetValueStringUnicode](#)

ziAPISyncSetValueD

ZIResult_enum ziAPISyncSetValueD (**ZIConnection** conn, const char* path, ZIDoubleData* value)

synchronously sets a double-type value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set to value

[in] value

Pointer to a double-type containing the value to be written. When the function returns value holds the effectively written value.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueD](#), [ziAPISetValueD](#)

ziAPISyncSetValueI

ZIResult_enum ziAPISyncSetValueI (**ZIConnection** conn, const char* path, **ZIIntegerData*** value)

synchronously sets an integer-type value to one or more nodes specified in a path

This function sets the values of the nodes specified in path to value. More than one value can be set if a wildcard is used. The function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the node(s) for which the value(s) will be set

[in] value

Pointer to a int-type containing then value to be written. when the function returns value holds the effectively written value.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueI](#), [ziAPISetValueI](#)

ziAPISyncSetValueB

ZIResult_enum ziAPISyncSetValueB ([ZIConnection](#) conn, const char* path, uint8_t* buffer, uint32_t* length, uint32_t bufferSize)

Synchronously sets the binary-type value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the [ziConnection](#) for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] buffer

Pointer to the byte array with the data

[in] length

Length of the data in the buffer

[in] bufferSize

Length of the data in the buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueB](#), [ziAPISetValueB](#)

ziAPISyncSetValueString

ZIResult_enum ziAPISyncSetValueString (**ZIConnection** conn, const char* path, const char* str)

Synchronously sets a string value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in/ out] str

Pointer to a null terminated string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueString](#), [ziAPISetValueString](#)

ziAPISyncSetValueStringUnicode

ZIResult_enum ziAPISyncSetValueStringUnicode (**ZIConnection** conn, const char* path, const wchar_t* wstr)

Synchronously sets a unicode string value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in/
out] wstr

Pointer to a null terminated unicode string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueStringUnicode](#), [ziAPISetValueStringUnicode](#)

ziAPISync

ZIResult_enum ziAPISync (**ZIConnection** conn)

Synchronizes the session by dropping all pending data.

This function drops any data that is pending for transfer. Any data (including poll data) retrieved afterwards is guaranteed to be produced not earlier than the call to ziAPISync. This ensures in particular that any settings made prior to the call to ziAPISync have been propagated to the device, and the data retrieved afterwards is produced with the new settings already set to the hardware. Note, however, that this does not include any required settling time.

Parameters:

[in] conn

Pointer to the ZIConnection that is to be synchronized

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIEchoDevice

ZIResult_enum ziAPIEchoDevice ([ZIConnection](#) conn, const char* deviceSerial)

Sends an echo command to a device and blocks until answer is received.

This is useful to flush all buffers between API and device to enforce that further code is only executed after the device executed a previous command. Per device echo is only implemented for HF2. For other device types it is a synonym to ziAPISync, and deviceSerial parameter is ignored.

Parameters:

[in] conn

Pointer to the ZIConnection that is to be synchronized

[in] deviceSerial

The serial of the device to get the echo from, e.g., dev2100

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIGetValueS

```
__inline ZIResult_enum ziAPIGetValueS ( ZIConnection conn, char* path,  
DemodSample* value )
```

ziAPIGetValueDIO

```
__inline ZIResult_enum ziAPIGetValueDIO ( ZIConnection conn, char* path,  
DIOSample* value )
```

ziAPIGetValueAuxIn

```
__inline ZIResult_enum ziAPIGetValueAuxIn ( ZIConnection conn, char* path,  
AuxInSample* value )
```

8.2.4. Data Streaming

This section describes how to perform data streaming. It allows for recording at high data rates without sample loss.

Data Structures

- struct [ZIEvent](#)
This struct holds event data forwarded by the Data Server.
- struct [ziEvent](#)
This struct holds event data forwarded by the Data Server.
Deprecated: See [ZIEvent](#).

Functions

- [ZIEvent*](#) [ziAPIAllocateEventEx](#) ()
Allocates [ZIEvent](#) structure and returns the pointer to it.
Attention!!! It is the client code responsibility to deallocate the structure by calling [ziAPIDeallocateEventEx](#)!
- void [ziAPIDeallocateEventEx](#) ([ZIEvent*](#) ev)
Deallocates [ZIEvent](#) structure created with [ziAPIAllocateEventEx\(\)](#).
- [ZIResult_enum](#) [ziAPISubscribe](#) ([ZIConnection](#) conn, const char* path)
subscribes the nodes given by path for [ziAPIPollDataEx](#)
- [ZIResult_enum](#) [ziAPIUnSubscribe](#) ([ZIConnection](#) conn, const char* path)
unsubscribes to the nodes given by path
- [ZIResult_enum](#) [ziAPIPollDataEx](#) ([ZIConnection](#) conn, [ZIEvent*](#) ev, uint32_t timeOutMilliseconds)
checks if an event is available to read
- [ZIResult_enum](#) [ziAPIGetValueAsPollData](#) ([ZIConnection](#) conn, const char* path)
triggers a value request, which will be given back on the poll event queue
- [__inline ZIResult_enum](#) [ziAPIPollData](#) ([ZIConnection](#) conn, [ziEvent*](#) ev, int timeOut)
Checks if an event is available to read. Deprecated: See [ziAPIPollDataEx\(\)](#).

Detailed Description

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>
#include <stdlib.h>

#include "ziAPI.h"

void EventLoop(ZIConnection Conn) {
    ZIResult_enum RetVal;
```

```
char* ErrBuffer;

ZIEvent* Event;
unsigned int Cnt = 0;

/*
   Allocate ZIEvent in heap memory instead of getting it from stack will
   secure against stack overflows especially in windows.
*/
if ((Event = ziAPIAllocateEventEx()) == NULL) {
    fprintf(stderr, "Can't allocate memory\n");
    return;
}

// Subscribe to a node, e.g., a demodulator sample.
if ((RetVal = ziAPISubscribe(Conn, "/dev1024/demod/0/sample")) != ZI_INFO_SUCCESS)
{
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't subscribe: %s\n", ErrBuffer);

    ziAPIDeallocateEventEx(Event);

    return;
}

// loop 1000 times
while (Cnt < 1000) {
    // get all demod rates from all devices every 10th cycle
    if (++Cnt % 10 == 0) {
        if ((RetVal =
            ziAPIGetValueAsPollData(
                Conn, "/dev1046/demods/*/rate")) != ZI_INFO_SUCCESS) {
            ziAPIGetError(RetVal, &ErrBuffer, NULL);
            fprintf(stderr, "Error, can't get value as poll data: %s.\n",
                ErrBuffer);

            break;
        }
    }
}

// Poll data until no more data is available.
while (1) {
    if ((RetVal = ziAPIPollDataEx(
        Conn, Event, 0)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't poll data: %s.\n", ErrBuffer);

        break;
    } else {
        // The field Count of the Event struct is zero when no data has been
        // polled
        if (Event->valueType != ZI_VALUE_TYPE_NONE && Event->count > 0) {
            /*
               process the received event here
            */
        } else {
            // no more data is available so go on
            break;
        }
    }
}

if (ziAPIUnsubscribe(Conn, "") != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't unsubscribe: %s.\n", ErrBuffer);
}
```

```
    ziAPIDeallocateEventEx(Event);  
}
```

Data Structure Documentation

struct ZIEvent

This struct holds event data forwarded by the Data Server.

```
#include "ziAPI.h"

typedef struct ZIEvent {
    uint32_t valueType;
    uint32_t count;
    uint8_t path[256];
    void* untyped;
    ZIDoubleData* doubleData;
    ZIDoubleDataTS* doubleDataTS;
    ZIIntegerData* integerData;
    ZIIntegerDataTS* integerDataTS;
    ZIComplexData* complexData;
    ZIByteArray* byteArray;
    ZIByteArrayTS* byteArrayTS;
    ZICntSample* cntSample;
    ZITrigSample* trigSample;
    ZITreeChangeData* treeChangeData;
    TreeChange* treeChangeDataOld;
    ZIDemodSample* demodSample;
    ZIAuxInSample* auxInSample;
    ZIDIOSample* dioSample;
    ZIScopeWave* scopeWave;
    ZIScopeWaveEx* scopeWaveEx;
    ScopeWave* scopeWaveOld;
    ZIPWAWave* pwaWave;
    ZISweeperWave* sweeperWave;
    ZISpectrumWave* spectrumWave;
    ZIAdvisorWave* advisorWave;
    ZIASyncReply* asyncReply;
    ZIVectorData* vectorData;
    ZIImpedanceSample* impedanceSample;
    uint64_t alignment;
    union ZIEvent::@6 value;
    uint8_t data[0x400000];
} ZIEvent;
```

Data Fields

- `uint32_t valueType`
Specifies the type of the data held by the [ZIEvent](#), see [ZIValueType_enum](#).
- `uint32_t count`
Number of values available in this event.
- `uint8_t path`
The path to the node from which the event originates.
- `void* untyped`
For convenience. The void field doesn't have a corresponding data type.
- `ZIDoubleData* doubleData`
when `valueType == ZI_VALUE_TYPE_DOUBLE_DATA`
- [ZIDoubleDataTS* doubleDataTS](#)

- when valueType == ZI_VALUE_TYPE_DOUBLE_DATA_TS
- [ZIntegerData*](#) integerData
when valueType == ZI_VALUE_TYPE_INTEGER_DATA
- [ZIntegerDataTS*](#) integerDataTS
when valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS
- [ZComplexData*](#) complexData
when valueType == ZI_VALUE_TYPE_COMPLEX_DATA
- [ZByteArray*](#) byteArray
when valueType == ZI_VALUE_TYPE_BYTE_ARRAY
- [ZByteArrayTS*](#) byteArrayTS
when valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS
- [ZCntSample*](#) cntSample
when valueType == ZI_VALUE_TYPE_CNT_SAMPLE
- [ZTrigSample*](#) trigSample
when valueType == ZI_VALUE_TYPE_TRIG_SAMPLE
- [ZTreeChangeData*](#) treeChangeData
when valueType == ZI_VALUE_TYPE_TREE_CHANGE_DATA
- [TreeChange*](#) treeChangeDataOld
when valueType ==
ZI_VALUE_TYPE_TREE_CHANGE_DATA_OLD
- [ZIDemodSample*](#) demodSample
when valueType == ZI_VALUE_TYPE_DEMOD_SAMPLE
- [ZAuxInSample*](#) auxInSample
when valueType == ZI_VALUE_TYPE_AUXIN_SAMPLE
- [ZIDIOSample*](#) dioSample
when valueType == ZI_VALUE_TYPE_DIO_SAMPLE
- [ZIScopeWave*](#) scopeWave
when valueType == ZI_VALUE_TYPE_SCOPE_WAVE
- [ZIScopeWaveEx*](#) scopeWaveEx
when valueType == ZI_VALUE_TYPE_SCOPE_WAVE_EX
- [ScopeWave*](#) scopeWaveOld
when valueType == ZI_VALUE_TYPE_SCOPE_WAVE_OLD
- [ZIPWAWave*](#) pwaWave
when valueType == ZI_VALUE_TYPE_PWA_WAVE
- [ZISweeperWave*](#) sweeperWave
when valueType == ZI_VALUE_TYPE_SWEEPER_WAVE
- [ZISpectrumWave*](#) spectrumWave

- when valueType == ZI_VALUE_TYPE_SPECTRUM_WAVE
- ZIAdvisorWave* advisorWave
when valueType == ZI_VALUE_TYPE_ADVISOR_WAVE
- ZIAsyncReply* asyncReply
when valueType == ZI_VALUE_TYPE_ASYNC_REPLY
- ZIVectorData* vectorData
when valueType == ZI_VALUE_TYPE_VECTOR_DATA
- ZIImpedanceSample* impedanceSample
when valueType == ZI_VALUE_TYPE_IMPEDANCE_SAMPLE
- uint64_t alignment
ensure union size is 8 bytes
- union ZIEvent::@6 value
Convenience pointer to allow for access to the first entry in Data using the correct type according to [ZIEvent.valueType](#) field.
- uint8_t data
The raw value data.

Detailed Description

[ZIEvent](#) is used to give out events like value changes or errors to the user. Event handling functionality is provided by [ziAPISubscribe](#) and [ziAPIUnSubscribe](#) as well as [ziAPIPollDataEx](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZIEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", Event->value.doubleData[j]);

        break;

    case ZI_VALUE_TYPE_INTEGER_DATA:

        printf("%u elements of integer data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", (float)Event->value.integerData[j]);

        break;
    }
```

```
case ZI_VALUE_TYPE_DEMOD_SAMPLE:

    printf("%u elements of sample data %s\n",
           Event->count,
           Event->path);

    for (j = 0; j < Event->count; j++)
        printf("TS=%f, X=%f, Y=%f.\n",
               (float)Event->value.demodSample[j].timeStamp,
               Event->value.demodSample[j].x,
               Event->value.demodSample[j].y);

    break;

case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

    printf("%u elements of tree-changed data, %s.\n",
           Event->count,
           Event->path);

    for (j = 0; j < Event->count; j++) {
        switch (Event->value.treeChangeDataOld[j].Action) {
            case ZI_TREE_ACTION_REMOVE:
                printf("Tree removed: %s\n",
                       Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_ADD:
                printf("treeChangeDataOld added: %s.\n",
                       Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_CHANGE:
                printf("treeChangeDataOld changed: %s.\n",
                       Event->value.treeChangeDataOld[j].Name);
                break;
        }
    }

    break;

default:

    printf("Unexpected event value type: %d.\n", Event->valueType);
    break;
}
}
```

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

struct `ziEvent`

This struct holds event data forwarded by the Data Server. Deprecated: See [ZIEvent](#).

```
#include "ziAPI.h"

typedef struct ziEvent {
    uint32_t Type;
    uint32_t Count;
    unsigned char Path[256];
    union ziEvent::Val Val;
    unsigned char Data[0x400000];
} ziEvent;
```

Data Structures

- union `ziEvent::Val`

Data Fields

- `uint32_t` Type
- `uint32_t` Count
- unsigned char Path
- union `ziEvent::Val` Val
- unsigned char Data

Detailed Description

[ziEvent](#) is used to give out events like value changes or errors to the user. Event handling functionality is provided by `ziAPISubscribe` and `ziAPIUnSubscribe` as well as `ziAPIPollDataEx`.

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZIEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", Event->value.doubleData[j]);
    }
```

```
        break;

    case ZI_VALUE_TYPE_INTEGER_DATA:

        printf("%u elements of integer data: %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", (float)Event->value.integerData[j]);

        break;

    case ZI_VALUE_TYPE_DEMOD_SAMPLE:

        printf("%u elements of sample data %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++)
            printf("TS=%f, X=%f, Y=%f.\n",
                   (float)Event->value.demodSample[j].timeStamp,
                   Event->value.demodSample[j].x,
                   Event->value.demodSample[j].y);

        break;

    case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

        printf("%u elements of tree-changed data, %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++) {
            switch (Event->value.treeChangeDataOld[j].Action) {
                case ZI_TREE_ACTION_REMOVE:
                    printf("Tree removed: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;

                case ZI_TREE_ACTION_ADD:
                    printf("treeChangeDataOld added: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;

                case ZI_TREE_ACTION_CHANGE:
                    printf("treeChangeDataOld changed: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;
            }
        }

        break;

    default:

        printf("Unexpected event value type: %d.\n", Event->valueType);
        break;
    }
}
```

Data Structure Documentation

union ziEvent::Val

```
typedef union ziEvent::Val {  
    void* Void;  
    DemodSample* SampleDemod;  
    AuxInSample* SampleAuxIn;  
    DIOSample* SampleDIO;  
    ziDoubleType* Double;  
    ziIntegerType* Integer;  
    TreeChange* Tree;  
    ByteArrayData* ByteArray;  
    ScopeWave* Wave;  
    uint64_t alignment;  
} ziEvent::Val;
```

Data Fields

- void* Void
- DemodSample* SampleDemod
- AuxInSample* SampleAuxIn
- DIOSample* SampleDIO
- ziDoubleType* Double
- ziIntegerType* Integer
- TreeChange* Tree
- ByteArrayData* ByteArray
- ScopeWave* Wave
- uint64_t alignment

Function Documentation

ziAPIAllocateEventEx

ZIEvent* ziAPIAllocateEventEx ()

Allocates [ZIEvent](#) structure and returns the pointer to it. Attention!!! It is the client code responsibility to deallocate the structure by calling [ziAPIDeallocateEventEx](#)!

This function allocates a [ZIEvent](#) structure and returns the pointer to it. Free the memory using [ziAPIDeallocateEventEx](#).

See Also:

[ziAPIDeallocateEventEx](#)

ziAPIDeallocateEventEx

void ziAPIDeallocateEventEx ([ZIEvent*](#) ev)

Deallocates [ZIEvent](#) structure created with [ziAPIAllocateEventEx\(\)](#).

Parameters:

[in] ev
Pointer to [ZIEvent](#) structure to be deallocated..

See Also:

[ziAPIAllocateEventEx](#)

This function is the compliment to [ziAPIAllocateEventEx\(\)](#)

ziAPISubscribe

ZIResult_enum ziAPISubscribe (**ZIConnection** conn, const char* path)

subscribes the nodes given by path for [ziAPIPollDataEx](#)

This function subscribes to nodes so that whenever the value of the node changes the new value can be polled using [ziAPIPollDataEx](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one leaf can be subscribed to with one function call.

Parameters:

[in] conn

Pointer to the ziConnection for which to subscribe for

[in] path

Path to the nodes to subscribe

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPIUnSubscribe](#), [ziAPIPollDataEx](#), [ziAPIGetValueAsPollData](#)

ziAPIUnSubscribe

ZIResult_enum ziAPIUnSubscribe (**ZIConnection** conn, const char* path)

unsubscribes to the nodes given by path

This function is the complement to [ziAPISubscribe](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one node can be unsubscribed with one function call.

Parameters:

[in] conn

Pointer to the ziConnection for which to unsubscribe for

[in] path

Path to the Nodes to unsubscribe

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIPollDataEx](#), [ziAPIGetValueAsPollData](#)

ziAPIPollDataEx

ZIResult_enum ziAPIPollDataEx (**ZIConnection** conn, **ZIEvent*** ev, uint32_t timeoutMilliseconds)

checks if an event is available to read

This function returns immediately if an event is pending. Otherwise it waits for an event for up to timeoutMilliseconds. All value changes that occur in nodes that have been subscribed to or in children of nodes that have been subscribed to are sent from the Data Server to the ziAPI session. For a description of how the data are available in the struct, refer to the documentation of struct [ziEvent](#). When no event was available within timeoutMilliseconds, the ziEvent::Type field will be ZI_DATA_NONE and the ziEvent::Count field will be zero. Otherwise these fields hold the values corresponding to the event that occurred.

Parameters:

[in] conn

Pointer to the [ZIConnection](#) for which events should be received

[out] ev

Pointer to a [ZIEvent](#) struct in which the received event will be written

[in] timeoutMilliseconds

Time to wait for an event in milliseconds. If -1 it will wait forever, if 0 the function returns immediately.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIGetValueAsPollData](#), [ziEvent](#)

ziAPIGetValueAsPollData

ZIResult_enum ziAPIGetValueAsPollData ([ZIConnection](#) conn, const char* path)

triggers a value request, which will be given back on the poll event queue

Use this function to receive the value of one or more nodes as one or more events using [ziAPIPollDataEx](#), even when the node is not subscribed or no value change has occurred.

Parameters:

[in] conn

Pointer to the [ZIConnection](#) with which the value should be retrieved

[in] path

Path to the Node holding the value

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

ziAPIPollData

`__inline ZIResult\_enum ziAPIPollData ( ZIConnection conn, ziEvent\* ev, int timeOut )`

Checks if an event is available to read. Deprecated: See [ziAPIPollDataEx\(\)](#).

Parameters:

[in] conn

Pointer to the [ZIConnection](#) for which events should be received

[out] ev

Pointer to a [ziEvent](#) struct in which the received event will be written

[in] timeOut

Time to wait for an event in milliseconds. If -1 it will wait forever, if 0 the function returns immediately.

Returns:

- [ZI_SUCCESS](#) On success.
- [ZI_CONNECTION](#) When the connection is invalid (not connected) or when a communication error occurred.
- [ZI_OVERFLOW](#) When a FIFO overflow occurred.

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIGetValueAsPollData](#), [ziEvent](#)

8.2.5. API for fast asynchronous operation

Functions in this group are non-blocking, and on return only report errors that can be identified directly on a client side (e.g. not connected). Any further results (including errors like node not found) of the command processing is returned as a special event in poll data. Tags are used to match the asynchronous replies with the sent commands.

Functions

- [ZIResult_enum](#) `ziAPIAsyncSetDoubleData ( ZIConnection conn, const char* path, ZIDoubleData value )`
- [ZIResult_enum](#) `ziAPIAsyncSetIntegerData ( ZIConnection conn, const char* path, ZIntegerData value )`
- [ZIResult_enum](#) `ziAPIAsyncSetByteArray ( ZIConnection conn, const char* path, uint8_t* buffer, uint32_t length )`
- [ZIResult_enum](#) `ziAPIAsyncSetString ( ZIConnection conn, const char* path, const char* str )`
- [ZIResult_enum](#) `ziAPIAsyncSetStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
- [ZIResult_enum](#) `ziAPIAsyncSubscribe ( ZIConnection conn, const char* path, ZIAsyncTag tag )`
- [ZIResult_enum](#) `ziAPIAsyncUnSubscribe ( ZIConnection conn, const char* path, ZIAsyncTag tag )`
- [ZIResult_enum](#) `ziAPIAsyncGetValueAsPollData ( ZIConnection conn, const char* path, ZIAsyncTag tag )`

Function Documentation

ziAPIAsyncSetDoubleData

[ZIResult_enum](#) ziAPIAsyncSetDoubleData ([ZIConnection](#) conn, const char* path, ZIDoubleData value)

ziAPIAsyncSetIntegerData

ZIResult_enum ziAPIAsyncSetIntegerData (**ZIConnection** conn, const char* path, ZIIntegerData value)

ziAPIAsyncSetByteArray

ZIResult_enum ziAPIAsyncSetByteArray (**ZIConnection** conn, const char* path, uint8_t* buffer, uint32_t length)

ziAPIAsyncSetString

ZIResult_enum ziAPIAsyncSetString (**ZIConnection** conn, const char* path, const char* str)

ziAPIAsyncSetStringUnicode

[ZIResult_enum](#) ziAPIAsyncSetStringUnicode ([ZIConnection](#) conn, const char* path, const wchar_t* wstr)

ziAPIAsyncSubscribe

[ZIResult_enum](#) ziAPIAsyncSubscribe ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

ziAPIAsyncUnSubscribe

[ZIResult_enum](#) ziAPIAsyncUnSubscribe ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

ziAPIAsyncGetValueAsPollData

[ZIResult_enum](#) ziAPIAsyncGetValueAsPollData ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

8.2.6. Error Handling and Logging in the LabOne C API

This section describes how to get more information when an error occurs.

Functions

- `ZIResult_enum` `ziAPIGetError ( ZIResult_enum result, char** buffer, int* base )`
Returns a description and the severity for a `ZIResult_enum`.
- `ZIResult_enum` `ziAPIGetLastError ( ZIConnection conn, char* buffer, uint32_t bufferSize )`
Returns the message from the last error that occurred.
- `void` `ziAPISetDebugLevel ( int32_t debugLevel )`
Enable ziAPI's log and set the severity level of entries to be included in the log.
- `void` `ziAPIWriteDebugLog ( int32_t debugLevel, const char* message )`
Write a message to ziAPI's log with the specified severity.

Detailed Description

In general, two types of errors can occur when using ziAPI. The two types are distinguished by the origin of the error: Whether it occurred within ziAPI itself or whether it occurred internally in the Zurich Instruments Core library.

All ziAPI functions (apart from a very few exceptions) return an exit code `ZIResult_enum`, which will be non-zero if the function call was not entirely successful. If the error originated in ziAPI itself, the exit code describes precisely the type of error that occurred (in other words, the exit code is not `ZI_ERROR_GENERAL`). In this case the error message corresponding to the exit code can be obtained with the function `ziAPIGetError`.

However, if the error has occurred internally, the exit code will be `ZI_ERROR_GENERAL`. In this case, the exit code does not describe the type of error precisely, instead a detailed error message is available to the user which can be obtained with the function `ziAPIGetLastError`. The function `ziAPIGetLastError` may be used with any function that takes a `ZIConnection` as an input argument (with the exception of `ziAPIInit`, `ziAPIDestroy`, `ziAPIConnect`, `ziAPIConnectEx`) and is the recommended function to use, if applicable, otherwise `ziAPIGetError` should be used.

The function `ziAPIGetLastError` was introduced in LabOne 15.11 due to the availability of `ziCoreModules` in ziAPI - its not desirable in general to map every possible error to an exit code in ziAPI; what is more relevant is the associated error message.

In addition to these two functions, ziAPI's log can be very helpful whilst debugging ziAPI-based programs. The log is not enabled by default; it's enabled by specifying a logging level with `ziAPISetDebugLevel`.

Function Documentation

ziAPIGetError

ZIResult_enum ziAPIGetError (**ZIResult_enum** result, char** buffer, int* base)

Returns a description and the severity for a **ZIResult_enum**.

This function returns a static char pointer to a description string for the given **ZIResult_enum** error code. It also provides a parameter returning the severity (info, warning, error). If the given error code does not exist a description for an unknown error and the base for an error will be returned. If a description or the base is not needed NULL may be passed. In general, it's recommended to use **ziAPIGetLastError** instead to get detailed error messages.

Parameters:

[in] result

A **ZIResult_enum** for which the description or base will be returned

[out] buffer

A pointer to a char array to return the description. May be NULL if no description is needed.

[out] base

The severity for the provided Status parameter:

- ZI_INFO_BASE For infos.
- ZI_WARNING_BASE For warnings.
- ZI_ERROR_BASE For errors.

Returns:

- ZI_INFO_SUCCESS Upon success.

ziAPIGetLastError

ZIResult_enum ziAPIGetLastError (**ZIConnection** conn, char* buffer, uint32_t bufferSize)

Returns the message from the last error that occurred.

This function can be used to obtain the error message from the last error that occurred associated with the provided **ZIConnection**. If the last ziAPI call is successful, then the last error message returned by ziAPIGetError is empty. Only ziAPI function calls that take **ZIConnection** as an input argument influence the message returned by ziAPIGetLastError, if they do not take **ZIConnection** as an input argument the last error message will neither be reset to be empty or set to an error message (in the case of the error). There are some exceptions to this rule, ziAPIGetLastError can also not be used with **ziAPIInit**, **ziAPIConnect**, **ziAPIConnectEx** and **ziAPIDestroy**. Note, a call to ziAPIGetLastError will also reset the last error message to empty if its call was successful. Since the buffer is left unchanged in the case of an error occurring in the call to ziAPIGetLastError it is safest to initialize the buffer with a known value, for example, "ziAPIGetLastError was not successful".

Parameters:

[in] conn

The **ZIConnection** from which to get the error message.

[out] buffer

A pointer to a char array to return the message.

[in] bufferSize

The length of the provided buffer.

Returns:

- **ZI_INFO_SUCCESS** Upon success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred. In this case the provided buffer is left unchanged.
- **ZI_ERROR_LENGTH** If the message's length exceeds the provided bufferSize, the message is truncated and written to buffer.

ziAPISetDebugLevel

void ziAPISetDebugLevel (int32_t debugLevel)

Enable ziAPI's log and set the severity level of entries to be included in the log.

Calling this function enables ziAPI's log at the specified severity level. On Windows the logs can be found by navigating to the Zurich Instruments "Logs" folder entry in the Windows Start Menu: Programs -> Zurich Instruments -> LabOne Servers -> Logs. This will open an Explorer window displaying folders containing log files from various LabOne components, in particular, the `ziAPILog` folder contains logs from ziAPI. On Linux, the logs can be found at `/tmp/ziAPILog_USERNAME`, where "USERNAME" is the same as the output of the "whoami" command.

Parameters:

[in] debugLevel

An integer specifying the log's severity level:

- trace: 0,
- info: 1,
- debug: 2,
- warning: 3,
- error: 4,
- fatal: 5,
- status: 6.

See Also:

[ziAPIWriteDebugLog](#)

ziAPIWriteDebugLog

void ziAPIWriteDebugLog (int32_t debugLevel, const char* message)

Write a message to ziAPI's log with the specified severity.

This function may be used to write a message to ziAPI's log from client code to assist with debugging. Note, this function is only available if the implementation used in [ziAPIConnectEx](#) is "ziAPI_Core" (the default implementation). Also logging must be first enabled using [ziAPISetDebugLevel](#).

Parameters:

[in] debugLevel

An integer specifying the severity of the message to write in the log:

- trace: 0,
- info: 1,
- debug: 2,
- warning: 3,
- error: 4,
- fatal: 5,
- status: 6.

[in] message

A character array comprising of the message to be written.

See Also:

[ziAPISetDebugLevel](#)

8.2.7. Using ziCore Modules in the LabOne C API

This sections describes ziAPI's interface for working with ziCore Modules. Modules provide a high-level interface for performing common measurement tasks such as sweeping data (Sweeper Module) or recording bursts of when certain trigger criteria have been fulfilled (Software Trigger Module). For an introduction to working with Modules please see the "ziCore Modules" section in the LabOne Programming Manual: .

Data Structures

- struct [ZIChunkHeader](#)
Structure to hold generic chunk data header information.
- struct [ZIModuleEvent](#)
This struct holds data of a single chunk from module lookup.

Typedefs

- typedef [ZIModuleEventPtr](#)
The pointer to a Module's data chunk to read out, updated via [ziAPIModGetChunk](#).

Enumerations

- enum [ZIChunkHeaderFlags_enum](#)
{ ZI_CHUNK_HEADER_FLAG_FINISHED,
ZI_CHUNK_HEADER_FLAG_ROLLMODE,
ZI_CHUNK_HEADER_FLAG_DATALOSS,
ZI_CHUNK_HEADER_FLAG_VALID,
ZI_CHUNK_HEADER_FLAG_DATA,
ZI_CHUNK_HEADER_FLAG_DISPLAY,
ZI_CHUNK_HEADER_FLAG_FREQDOMAIN,
ZI_CHUNK_HEADER_FLAG_SPECTRUM,
ZI_CHUNK_HEADER_FLAG_OVERLAPPED,
ZI_CHUNK_HEADER_FLAG_ROWFINISHED,
ZI_CHUNK_HEADER_FLAG_ONGRIDSAMPLING,
ZI_CHUNK_HEADER_FLAG_ROWREPETITION,
ZI_CHUNK_HEADER_FLAG_PREVIEW }
Defines the flags returned in the chunk header for all modules.
- enum [ZIChunkHeaderModuleFlags_enum](#)
{ ZI_CHUNK_HEADER_MODULE_FLAGS_WINDOW }
Defines flags returned in the chunk header that only apply for certain modules.

Functions

- [ZIResult_enum](#) [ziAPIModCreate](#) ([ZIConnection](#) conn, [ZIModuleHandle*](#) handle, const char* moduleId)
Create a [ZIModuleHandle](#) that can be used for asynchronous measurement tasks.

- `ZIResult_enum` `ziAPIModSetDoubleData ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIDoubleData value )`
Sets a module parameter to the specified double type.
- `ZIResult_enum` `ziAPIModSetIntegerData ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData value )`
Sets a module parameter to the specified integer type.
- `ZIResult_enum` `ziAPIModSetByteArray ( ZIConnection conn, ZIModuleHandle handle, const char* path, uint8_t* buffer, uint32_t length )`
Sets a module parameter to the specified byte array.
- `ZIResult_enum` `ziAPIModSetString ( ZIConnection conn, ZIModuleHandle handle, const char* path, const char* str )`
Sets a module parameter to the specified null-terminated string.
- `ZIResult_enum` `ziAPIModSetStringUnicode ( ZIConnection conn, ZIModuleHandle handle, const char* path, const wchar_t* wstr )`
Sets a module parameter to the specified null-terminated unicode string.
- `ZIResult_enum` `ziAPIModGetInteger ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData* value )`
gets the integer-type value of the specified node
- `ZIResult_enum` `ziAPIModGetDouble ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIDoubleData* value )`
gets the double-type value of the specified node
- `ZIResult_enum` `ziAPIModGetString ( ZIConnection conn, ZIModuleHandle handle, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize )`
gets the null-terminated string value of the specified node
- `ZIResult_enum` `ziAPIModGetStringUnicode ( ZIConnection conn, ZIModuleHandle handle, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize )`
gets the null-terminated string value of the specified node
- `ZIResult_enum` `ziAPIModListNodes ( ZIConnection conn, ZIModuleHandle handle, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags )`
Returns all child parameter node paths found under the specified parent module parameter path.
- `ZIResult_enum` `ziAPIModSubscribe ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Subscribes to the nodes specified by path, these nodes will be recorded during module execution.

- `ZIResult_enum` `ziAPIModUnSubscribe ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Unsubscribes to the nodes specified by path.
- `ZIResult_enum` `ziAPIModExecute ( ZIConnection conn, ZIModuleHandle handle )`
Starts the module's thread and its associated measurement task.
- `ZIResult_enum` `ziAPIModTrigger ( ZIConnection conn, ZIModuleHandle handle )`
Manually issue a trigger forcing data recording (SW Trigger Module only).
- `ZIResult_enum` `ziAPIModProgress ( ZIConnection conn, ZIModuleHandle handle, ZIDoubleData* progress )`
Queries the current state of progress of the module's measurement task.
- `ZIResult_enum` `ziAPIModFinished ( ZIConnection conn, ZIModuleHandle handle, ZIntegerData* finished )`
Queries whether the module has finished its measurement task.
- `ZIResult_enum` `ziAPIModFinish ( ZIConnection conn, ZIModuleHandle handle )`
Stops the module performing its measurement task.
- `ZIResult_enum` `ziAPIModSave ( ZIConnection conn, ZIModuleHandle handle, const char* fileName )`
Saves the currently accumulated data to file.
- `ZIResult_enum` `ziAPIModRead ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Make the currently accumulated data available for use in the C program.
- `ZIResult_enum` `ziAPIModNextNode ( ZIConnection conn, ZIModuleHandle handle, char* path, uint32_t bufferSize, ZIValueType_enum* valueType, uint64_t* chunks )`
Make the data for the next node available for reading with `ziAPIModGetChunk`.
- `ZIResult_enum` `ziAPIModGetChunk ( ZIConnection conn, ZIModuleHandle handle, uint64_t chunkIndex, ZIModuleEventPtr* ev )`
Get the specified data chunk from the current node.
- `ZIResult_enum` `ziAPIModEventDeallocate ( ZIConnection conn, ZIModuleHandle handle, ZIModuleEventPtr ev )`
Deallocate the `ZIModuleEventPtr` being used by the module.
- `ZIResult_enum` `ziAPIModClear ( ZIConnection conn, ZIModuleHandle handle )`
Terminates the module's thread and destroys the module.

Data Structure Documentation

struct ZIChunkHeader

Structure to hold generic chunk data header information.

```
#include "ziAPI.h"

typedef struct ZIChunkHeader {
    ZITimeStamp systemTime;
    ZITimeStamp createdTimeStamp;
    ZITimeStamp changedTimeStamp;
    uint32_t flags;
    uint32_t moduleFlags;
    uint32_t status;
    uint32_t reserved0;
    uint64_t chunkSizeBytes;
    uint64_t triggerNumber;
    char name[32];
    uint32_t groupIndex;
    uint32_t color;
    uint32_t activeRow;
    uint32_t gridRows;
    uint32_t gridCols;
    uint32_t gridMode;
    uint32_t gridOperation;
    uint32_t gridDirection;
    uint32_t gridRepetitions;
    double gridColDelta;
    double gridColOffset;
    double gridRowDelta;
    double gridRowOffset;
    double bandwidth;
    double center;
    double nenbw;
} ZIChunkHeader;
```

Data Fields

- ZITimeStamp systemTime
System timestamp.
- ZITimeStamp createdTimeStamp
Creation timestamp.
- ZITimeStamp changedTimeStamp
Last changed timestamp.
- uint32_t flags
Flags (bitmask of values from ZIChunkHeaderFlags_enum)
- uint32_t moduleFlags
moduleFlags (bitmask of values from
ZIChunkHeaderModuleFlags_enum, module-specific)
- uint32_t status
Status Flag: [0] : selected [1] : group assigned [2] : color
edited [4] : name edited.
- uint32_t reserved0

- `uint64_t chunkSizeBytes`
Size in bytes used for memory usage calculation.
- `uint64_t triggerNumber`
SW Trigger counter since execution start.
- `char name`
Name in history list.
- `uint32_t groupIndex`
Group index in history list.
- `uint32_t color`
Color in history list.
- `uint32_t activeRow`
Active row in history list.
- `uint32_t gridRows`
Number of grid rows.
- `uint32_t gridCols`
Number of grid columns.
- `uint32_t gridMode`
Grid mode interpolation mode (0 = off, 1 = nearest, 2 = linear, 3 = Lanczos)
- `uint32_t gridOperation`
Grid mode operation (0 = replace, 1 = average)
- `uint32_t gridDirection`
Grid mode direction (0 = forward, 1 = revers, 2 = bidirectional)
- `uint32_t gridRepetitions`
Number of repetitions in grid mode.
- `double gridColDelta`
Delta between grid points in SI unit.
- `double gridColOffset`
Offset of first grid point relative to trigger.
- `double gridRowDelta`
Delta between grid rows in SI unit.
- `double gridRowOffset`
Delay of first grid row relative to trigger.
- `double bandwidth`
For FFT the bandwidth of the signal.
- `double center`
The FFT center frequency.
- `double nenbw`

For FFT the normalized effective noise bandwidth.

struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"

typedef struct ZIModuleEvent {
    uint64_t allocatedSize;
    ZIChunkHeader* header;
    ZIEvent
        value[0];
} ZIModuleEvent;
```

Data Fields

- `uint64_t allocatedSize`
For internal use - never modify!
- `ZIChunkHeader* header`
Chunk header.
- `ZIEvent value`
Defines location of stored `ZIEvent`.

Enumeration Type Documentation

Defines the flags returned in the chunk header for all modules.

Enumerator:

- `ZI_CHUNK_HEADER_FLAG_FINISHED`
Indicates that the chunk data is complete. This flag will be set if data is read out from the module before the measurement (e.g. sweep) has finished.
- `ZI_CHUNK_HEADER_FLAG_ROLLMODE`
Unused.
- `ZI_CHUNK_HEADER_FLAG_DATALOSS`
Indicates that dataloss has occurred.
- `ZI_CHUNK_HEADER_FLAG_VALID`
Indicates that the data is valid.
- `ZI_CHUNK_HEADER_FLAG_DATA`
Indicates whether the chunk contains data (opposite to setting).
- `ZI_CHUNK_HEADER_FLAG_DISPLAY`
Internal use only.
- `ZI_CHUNK_HEADER_FLAG_FREQDOMAIN`
chunk contains frequency domain data, opposite to time domain.
- `ZI_CHUNK_HEADER_FLAG_SPECTRUM`
chunk recorded in spectrum mode.
- `ZI_CHUNK_HEADER_FLAG_OVERLAPPED`
chunk results overlap with neighboring chunks, see spectrum.
- `ZI_CHUNK_HEADER_FLAG_ROWFINISHED`
indicates that the current row is finished - useful for row first averaging.
- `ZI_CHUNK_HEADER_FLAG_ONGRIDSAMPLING`
exact sampling was used.
- `ZI_CHUNK_HEADER_FLAG_ROWREPETITION`
row first averaging is enabled.
- `ZI_CHUNK_HEADER_FLAG_PREVIEW`
chunk contains preview (fft with less points).

Defines flags returned in the chunk header that only apply for certain modules.

Enumerator:

- ZI_CHUNK_HEADER_MODULE_FLAGS_WINDOW
FFT Window used in the Data Acquisition Module: 0 - Rectangular, 1 - Hann, 2 - Hamming, 3 - Blackmanharris, 4 - Exponential, 5 - Cosine, 6 - CosineSquare.

Function Documentation

ziAPIModCreate

ZIResult_enum ziAPIModCreate (**ZIConnection** conn, **ZIModuleHandle*** handle, const char* moduleId)

Create a **ZIModuleHandle** that can be used for asynchronous measurement tasks.

This function initializes a ziCore module and provides a pointer (handle) with which to access and work with it. Note that this function does not start the module's thread. Before the thread can be started (with **ziAPIModExecute**):

- the device serial (e.g., "dev100") to be used with module must be specified via **ziAPIModSetByteArray**.
- the desired data (node paths) to record during the measurement must be specified via **ziAPIModSubscribe**. The module's thread is stopped with **ziAPIModClear**.

Parameters:

[in] conn

The **ZIConnection** which should be used to initialize the module.

[out] handle

Pointer to the initialized **ZIModuleHandle**, which from then on can be used to reference the module.

[in] moduleId

The name specifying the type the module to create (only the following ziCore Modules are currently supported in ziAPI):

- "sweep" to initialize an instance of the Sweeper Module.
- "record" to initialize an instance of the Software Trigger (Recorder) Module.
- "zoomFFT" to initialize an instance of the Spectrum Module.
- "deviceSettings" to initialize an instance to save/load device settings.
- "pidAdvisor" to initialize an instance of the PID Advisor Module.
- "awgModule" to initialize an instance of the AWG Compiler Module.
- "impedanceModule" to initialize an instance of the Impedance Compensation Module.
- "scopeModule" to initialize an instance of the Scope Module to assembly scope shots.
- "multiDeviceSyncModule" to initialize an instance of the Device Synchronization Module.
- "dataAcquisitionModule" to initialize an instance of the Data Acquisition Module.

Returns:

- ZI_INFO_SUCCESS On success.

- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_WARNING_NOTFOUND if the provided moduleId was invalid.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModClear](#)

ziAPIModSetDoubleData

ZIResult_enum ziAPIModSetDoubleData (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIDoubleData** value)

Sets a module parameter to the specified double type.

This function is used to configure (set) module parameters which have double types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] value

The double data to write to the path.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#), [ziAPIModSetString](#)

ziAPIModSetIntegerData

ZIResult_enum ziAPIModSetIntegerData (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIntegerData** value)

Sets a module parameter to the specified integer type.

This function is used to configure (set) module parameters which have integer types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] value

The integer data to write to the path.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetByteArray](#), [ziAPIModSetString](#)

ziAPIModSetByteArray

ZIResult_enum ziAPIModSetByteArray ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* path, uint8_t* buffer, uint32_t length)

Sets a module parameter to the specified byte array.

This function is used to configure (set) module parameters which have byte array types.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] buffer

Pointer to the byte array with the data.

[in] length

Length of the data in the buffer.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetString](#)

ziAPIModSetString

ZIResult_enum ziAPIModSetString (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, const char* str)

Sets a module parameter to the specified null-terminated string.

This function is used to configure (set) module parameters which have string types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] str

Pointer to a null-terminated string (max 64k characters).

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#)

ziAPIModSetStringUnicode

ZIResult_enum ziAPIModSetStringUnicode ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* path, const wchar_t* wstr)

Sets a module parameter to the specified null-terminated unicode string.

This function is used to configure (set) module parameters which have string types.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] wstr

Pointer to a null-terminated unicode string (max 64k characters).

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#)

ziAPIModGetInteger

ZIResult_enum ziAPIModGetInteger (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIntegerData*** value)

gets the integer-type value of the specified node

This function is used to retrieve module parameter values of type integer.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the node holding the value

[out] value

Pointer to an 64bit integer in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetDouble](#), [ziApiModGetString](#)

ziAPIModGetDouble

ZIResult_enum ziAPIModGetDouble (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIDoubleData*** value)

gets the double-type value of the specified node

This function is used to retrieve module parameter values of type floating point double.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the node holding the value

[out] value

Pointer to an floating point double in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetString](#)

ziAPIModGetString

ZIResult_enum ziAPIModGetString (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize)

gets the null-terminated string value of the specified node

This function is used to retrieve module parameter values of type string.

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetDouble](#)

ziAPIModGetStringUnicode

ZIResult_enum ziAPIModGetStringUnicode (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize)

gets the null-terminated string value of the specified node

This function is used to retrieve module parameter values of type string.

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the Node holding the value

[out] wbuffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetDouble](#), [ziAPIModGetString](#)

ziAPIModListNodes

ZIResult_enum ziAPIModListNodes (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child parameter node paths found under the specified parent module parameter path.

This function returns a list of parameter names found at the specified path. The path may contain wildcards. The list is returned in a null-terminated char-buffer, each element delimited by a newline. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be **ZI_ERROR_LENGTH**. Note, the provided path must match the module being addressed, i.e., path must exactly start with "sweep/" for the Sweeper Module.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** from which the parameter names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with newline-delimited list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer specified as the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in **ZIListNodes_enum**.

Returns:

- **ZI_INFO_SUCCESS** On success
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_LENGTH** If the path's length exceeds **MAX_PATH_LEN** or the length of the char-buffer for the nodes given by bufferSize is too small for all elements.
- **ZI_WARNING_OVERFLOW** When a FIFO overflow occurred.
- **ZI_ERROR_COMMAND** On an incorrect answer of the server.
- **ZI_ERROR_SERVER_INTERNAL** If an internal error occurred in Data Server.
- **ZI_WARNING_NOTFOUND** If the given path could not be resolved.
- **ZI_ERROR_TIMEOUT** When communication timed out.

- `ZI_ERROR_GENERAL` If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIModSubscribe

ZIResult_enum ziAPIModSubscribe (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path)

Subscribes to the nodes specified by path, these nodes will be recorded during module execution.

This function subscribes to nodes so that whenever the value of the node changes while the module is executing the new value will be accumulated and then read using [ziAPIModRead](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one leaf can be subscribed to with one function call.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module in which the nodes should be subscribed to.

[in] path

Path specifying the nodes to subscribe to, may contain wildcards.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or a general error occurred, enable ziAPI's log for detailed information, see [ziAPISetDebugLevel](#).
- ZI_ERROR_LENGTH If the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW When a FIFO overflow occurred.
- ZI_ERROR_COMMAND On an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL If an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND If the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT When communication timed out.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModUnSubscribe](#), [ziAPIModRead](#)

ziAPIModUnSubscribe

ZIResult_enum ziAPIModUnSubscribe (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path)

Unsubscribes to the nodes specified by path.

This function is the complement to [ziAPIModSubscribe](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one node can be unsubscribed with one function call.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module in which the nodes should be unsubscribed from.

[in] path

Path specifying the nodes to unsubscribe from, may contain wildcards.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH If the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW When a FIFO overflow occurred.
- ZI_ERROR_COMMAND On an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL If an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND If the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT When communication timed out.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSubscribe](#), [ziAPIModRead](#)

ziAPIModExecute

ZIResult_enum `ziAPIModExecute ( ZIConnection conn, ZIModuleHandle handle )`

Starts the module's thread and its associated measurement task.

Once the module's parameters has been configured as required via, e.g. [ziAPIModSetDoubleData](#), this function starts the module's thread. This starts the module's main measurement task which will run asynchronously. The thread will run until either the module has completed its task or until [ziAPIModFinish](#) is called. Subscription or unsubscription is not possible while the module is executing. The status of the module can be obtained with either [ziAPIModFinished](#) or [ziAPIModProgress](#).

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModCreate](#), [ziAPIModProgress](#), [ziAPIModFinish](#)

ziAPIModTrigger

ZIResult_enum ziAPIModTrigger ([ZIConnection](#) conn, [ZIModuleHandle](#) handle)

Manually issue a trigger forcing data recording (SW Trigger Module only).

This function is used with the Software Trigger Module in order to manually issue a trigger in order to force recording of data. A burst of subscribed data will be recorded as configured via the SW Trigger's parameters as would a regular trigger event.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIModProgress

ZIResult_enum ziAPIModProgress (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIDoubleData*** progress)

Queries the current state of progress of the module's measurement task.

This function can be used to query the module's progress in performing its current measurement task, the progress is returned as a double in [0, 1], where 1 indicates task completion.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] progress

A pointer to **ZIDoubleData** indicating the current progress of the module.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

ziAPIModExecute, **ziAPIModFinish**, **ziAPIModFinished**

ziAPIModFinished

ZIResult_enum ziAPIModFinished (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIIntegerData*** finished)

Queries whether the module has finished its measurement task.

This function can be used to query whether the module has finished its task or not.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] finished

A pointer to **ZIIntegerData**, upon return this will be 0 if the module is still executing or 1 if it has finished executing.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#), [ziAPIModProgress](#)

ziAPIModFinish

ZIResult_enum ziAPIModFinish (**ZIConnection** conn, **ZIModuleHandle** handle)

Stops the module performing its measurement task.

This function stops the module performing its associated measurement task and stops recording any data. The task and data recording may be restarted by calling [ziAPIModExecute](#) again.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModProgress](#), [ziAPIModFinished](#)

ziAPIModSave

ZIResult_enum ziAPIModSave ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* fileName)

Saves the currently accumulated data to file.

This function saves the currently accumulated data to a file. The path of the file to save data to is specified via the module's directory parameter.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[in] fileName

The basename of the file to save the data in.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#), [ziAPIModFinished](#)

ziAPIModRead

ZIResult_enum ziAPIModRead ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* path)

Make the currently accumulated data available for use in the C program.

This function can be used to either read (get) module parameters, in this case a path that addresses the module must be specified, or it can be used to read out the currently accumulated data from subscribed nodes in the module. In either case the actual data must then be accessed by the user using [ziAPIModNextNode](#) and [ziAPIModGetChunk](#).

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[in] path

The path specifying the module parameter(s) to get, specify NULL to obtain all subscribed data.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetChunk](#), [ziAPIModNextNode](#)

ziAPIModNextNode

ZIResult_enum ziAPIModNextNode (**ZIConnection** conn, **ZIModuleHandle** handle, char* path, uint32_t bufferSize, **ZIValueType_enum*** valueType, uint64_t* chunks)

Make the data for the next node available for reading with [ziAPIModGetChunk](#).

After calling [ziAPIModRead](#), subscribed data (or module parameters) may now be read out on a node-by-node and chunk-by-chunk basis. All nodes with data available in the module can be iterated over by using [ziAPIModNextNode](#), then for each node the chunks of data available are read out using [ziAPIModGetChunk](#). Calling this function makes the data from the next node available for read.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[out] path

A string specifying the node's path whose data chunk points to.

[in] bufferSize

The length of the buffer specified as the path output parameter.

[out] valueType

The [ZIValueType_enum](#) of the node's data.

[out] chunks

The number of chunks of data available for the node.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModRead](#), [ziAPIModGetChunk](#), [ziAPIModEventDeallocate](#)

ziAPIModGetChunk

ZIResult_enum ziAPIModGetChunk (**ZIConnection** conn, **ZIModuleHandle** handle, **uint64_t** chunkIndex, **ZIModuleEventPtr*** ev)

Get the specified data chunk from the current node.

Data is read out node-by-node and then chunk-by-chunk. This function can be used to obtain specific data chunks from the current node that data is being read from. More precisely, it preallocates space for an event structure big enough to hold the node's data at the specified chunk index, updates **ZIModuleEventPtr** to point to this space and then copies the chunk data to this space.

Note, before the very first call to ziAPIModGetChunk, the **ZIModuleEventPtr** should be initialized to NULL and then left untouched for all subsequent calls (even after calling **ziAPIModNextNode** to get data from the next node). This is because ziAPIModGetChunk internally manages the required space allocation for the event and then in subsequent calls only reallocates space when it is required. It is optimized to reduce the number of required space reallocations for the event.

The **ZIModuleEventPtr** should be deallocated using **ziAPIModEventDeallocate**, otherwise the lifetime of the **ZIModuleEventPtr** is the same as the lifetime of the module. Indeed, the same **ZIModuleEventPtr** can be used, even for subsequent reads. It is also possible to work with multiple **ZIModuleEventPtr** so that some pointers can be kept for later processing.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] chunkIndex

The index of the data chunk to update the pointer to.

[out] ev

The module's **ZIModuleEventPtr** that points to the currently available data chunk.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

[ziAPIModRead](#), [ziAPIModNextNode](#), [ziAPIModEventDeallocate](#)

ziAPIModEventDeallocate

ZIResult_enum ziAPIModEventDeallocate (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIModuleEventPtr** ev)

Deallocate the **ZIModuleEventPtr** being used by the module.

This function deallocates the **ZIModuleEventPtr**. Since a module event's allocated space is managed internally by **ziAPIModGetChunk**, when the user no longer requires the event (all data has been read out) it must be deallocated by the user with this function.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[in] ev

The **ZIModuleEventPtr** to deallocate.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

ziAPIModGetChunk, **ziAPIModRead**

ziAPIModClear

ZIResult_enum ziAPIModClear ([ZIConnection](#) conn, [ZIModuleHandle](#) handle)

Terminates the module's thread and destroys the module.

This function terminates the module's thread. After calling ziAPIModClear the module's handle may not be used any more. A new instance of the module must be initialized if required.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#)

8.2.8. Vector Write

Functions for working with vector data writing.

Enumerations

- enum [ZIVectorWriteStatus_enum](#)
{ ZI_VECTOR_WRITE_STATUS_IDLE,
ZI_VECTOR_WRITE_STATUS_PENDING }

Functions

- [ZIResult_enum](#) ziAPIVectorWriteBlock ([ZIConnection](#) conn,
const char* path, [ZIVectorData*](#) vectorBlock)
- [ZIResult_enum](#) ziAPIVectorWriteGetStatus ([ZIConnection](#)
conn, const char* path, uint8_t* status)
status - see [::ZIVectorWriteStatus_enum](#)
- [ZIResult_enum](#) ziAPIVectorWrite ([ZIConnection](#) conn, const
char* path, const void* vectorPtr, uint8_t vectorElementType,
uint64_t vectorSizeElements)
vectorElementType - see [ZIVectorElementType_enum](#)

Enumeration Type Documentation

Enumerator:

- ZI_VECTOR_WRITE_STATUS_IDLE
- ZI_VECTOR_WRITE_STATUS_PENDING

Function Documentation

ziAPIVectorWriteBlock

[ZIResult_enum](#) ziAPIVectorWriteBlock ([ZIConnection](#) conn, const char* path, [ZIVectorData*](#) vectorBlock)

ziAPIVectorWriteGetStatus

[ZIResult_enum](#) ziAPIVectorWriteGetStatus ([ZIConnection](#) conn, const char* path, uint8_t* status)

status - see ::ZIVectorWriteStatus_enum

ziAPIVectorWrite

[ZIResult_enum](#) ziAPIVectorWrite ([ZIConnection](#) conn, const char* path, const void* vectorPtr, uint8_t vectorElementType, uint64_t vectorSizeElements)

vectorElementType - see [ZIVectorElementType_enum](#)

8.2.9. Device discovery

Functions for working with device Discovery.

Functions

- [ZIResult_enum](#) `ziAPIDiscoveryFindAll ( ZIConnection conn, char* devicelds, uint32_t bufferSize )`
- [ZIResult_enum](#) `ziAPIDiscoveryFind ( ZIConnection conn, const char* deviceAddress, const char** deviceld )`
- [ZIResult_enum](#) `ziAPIDiscoveryGet ( ZIConnection conn, const char* deviceld, const char** propsJSON )`
- [ZIResult_enum](#) `ziAPIDiscoveryGetValueI ( ZIConnection conn, const char* deviceld, const char* propName, ZIntegerData* value )`
- [ZIResult_enum](#) `ziAPIDiscoveryGetValueS ( ZIConnection conn, const char* deviceld, const char* propName, const char** value )`

Function Documentation

ziAPIDiscoveryFindAll

ZIResult_enum ziAPIDiscoveryFindAll (**ZIConnection** conn, char* deviceIds, uint32_t bufferSize)

Perform a Discovery property look-up for the specified deviceAddress and return its device ID. Attention! This invalidates all pointers previously returned by ziAPIDiscovery* calls. The deviceId need not be deallocated by the user.

Parameters:

[in] conn

Pointer to **ZIConnection** with which the value should be retrieved.

[out] deviceIds

Pointer to a buffer that is to contain the list of newline-separated IDs of the devices found, e.g. "DEV2006\nDEV2007\n".

[in] bufferSize

The size of the buffer pointed to by deviceIds. If the buffer is too small to hold the complete list of device IDs, its contents remain unchanged.

Returns:

- ZI_INFO_SUCCESS
- ZI_ERROR_LENGTH The provided buffer is too small to hold the list.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryFind

ZIResult_enum ziAPIDiscoveryFind (**ZIConnection** conn, const char* deviceAddress, const char** deviceId)

Perform a Discovery property look-up for the specified deviceAddress and return its device ID. Attention! This invalidates all pointers previously returned by ziAPIDiscovery* calls. The deviceId need not be deallocated by the user.

Parameters:

[in] conn

Pointer to **ZIConnection** with which the value should be retrieved.

[in] deviceAddress

The address or ID of the device to find, e.g., 'uhf-dev2006' or 'dev2006'.

[out] deviceId

The ID of the device that was found, e.g. 'DEV2006'.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFindAll](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGet

ZIResult_enum ziAPIDiscoveryGet (**ZIConnection** conn, const char* deviceId, const char** propsJSON)

Returns the device Discovery properties for a given device ID in JSON format. The function [ziAPIDiscoveryFind](#) must be called before ziAPIDiscoveryGet can be used. The propsJSON need not be deallocated by the user.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[out] propsJSON

The Discovery properites in JSON format of the specified device.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGetValueI

ZIResult_enum ziAPIDiscoveryGetValueI (**ZIConnection** conn, const char* deviceId, const char* propName, ZIIntegerData* value)

Returns the specified integer Discovery property value for a given device ID. The function [ziAPIDiscoveryFind](#) must be called with the required device ID before using ziAPIDiscoveryGetValueI.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[in] propName

The name of the desired integer Discovery property.

[out] value

Pointer to the value of the specified Discovery property.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGetValueS

ZIResult_enum ziAPIDiscoveryGetValueS (**ZIConnection** conn, const char* deviceId, const char* propName, const char** value)

Returns the specified string Discovery property value for a given device ID. The function [ziAPIDiscoveryFind](#) must be called with the required device ID before using ziAPIDiscoveryGetValueS. The value must not be deallocated by the user.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[in] propName

The name of the desired integer Discovery property.

[out] value

Pointer to the value of the specified Discovery property.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#)

8.3. Data Structure Documentation

8.3.1. struct AuxInSample

The [AuxInSample](#) struct holds data for the ZI_DATA_AUXINSAMPLE data type. Deprecated: See [ZIAuxInSample](#).

```
#include "ziAPI.h"

typedef struct AuxInSample {
    ziTimeStampType TimeStamp;
    double Ch0;
    double Ch1;
} AuxInSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `double Ch0`
- `double Ch1`

8.3.2. struct ByteArrayData

The [ByteArrayData](#) struct holds data for the ZI_DATA_BYTEARRAY data type. Deprecated: See [ZIByteArray](#).

```
#include "ziAPI.h"

typedef struct ByteArrayData {
    unsigned int Len;
    unsigned char Bytes[0];
} ByteArrayData;
```

Data Fields

- unsigned int Len
- unsigned char Bytes

8.3.3. struct DemodSample

The [DemodSample](#) struct holds data for the ZI_DATA_DEMODSAMPLE data type. Deprecated: See [ZIDemodSample](#).

```
#include "ziAPI.h"

typedef struct DemodSample {
    ziTimeStampType TimeStamp;
    double X;
    double Y;
    double Frequency;
    double Phase;
    unsigned int DIOBits;
    unsigned int Reserved;
    double AuxIn0;
    double AuxIn1;
} DemodSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `double X`
- `double Y`
- `double Frequency`
- `double Phase`
- `unsigned int DIOBits`
- `unsigned int Reserved`
- `double AuxIn0`
- `double AuxIn1`

8.3.4. struct DIOSample

The [DIOSample](#) struct holds data for the ZI_DATA_DIOSAMPLE data type. Deprecated: See [ZIDIOSample](#).

```
#include "ziAPI.h"

typedef struct DIOSample {
    ziTimeStampType TimeStamp;
    unsigned int Bits;
    unsigned int Reserved;
} DIOSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `unsigned int Bits`
- `unsigned int Reserved`

8.3.5. struct ScopeWave

The structure used to hold a single scope shot (API Level 1). If the client is connected to the Data Server using API Level 4 (recommended if supported by your device class) please see [ZIScopeWave](#) instead ([ZIScopeWaveEx](#) for API Level 5 and above).

```
#include "ziAPI.h"

typedef struct ScopeWave {
    double dt;
    uint32_t ScopeChannel;
    uint32_t TriggerChannel;
    uint32_t BWLimit;
    uint32_t Count;
    int16_t Data[0];
} ScopeWave;
```

Data Fields

- double dt
Time difference between samples.
- uint32_t ScopeChannel
Scope channel of the represented data.
- uint32_t TriggerChannel
Trigger channel of the represented data.
- uint32_t BWLimit
Bandwidth-limit flag.
- uint32_t Count
Count of samples.
- int16_t Data
First wave data.

8.3.6. struct TreeChange

The structure used to hold info about added or removed nodes. This is the version without timestamp used in API v1 compatibility mode.

```
#include "ziAPI.h"

typedef struct TreeChange {
    uint32_t Action;
    char Name[32];
} TreeChange;
```

Data Fields

- uint32_t Action
field indicating which action occurred on the tree. A value of the [ZITreeAction_enum](#) (TREE_ACTION) enum.
- char Name
Name of the Path that has been added, removed or changed.

8.3.7. union ziEvent::Val

```
typedef union ziEvent::Val {  
    void* Void;  
    DemodSample* SampleDemod;  
    AuxInSample* SampleAuxIn;  
    DIOSample* SampleDIO;  
    ziDoubleType* Double;  
    ziIntegerType* Integer;  
    TreeChange* Tree;  
    ByteArrayData* ByteArray;  
    ScopeWave* Wave;  
    uint64_t alignment;  
} ziEvent::Val;
```

Data Fields

- void* Void
- DemodSample* SampleDemod
- AuxInSample* SampleAuxIn
- DIOSample* SampleDIO
- ziDoubleType* Double
- ziIntegerType* Integer
- TreeChange* Tree
- ByteArrayData* ByteArray
- ScopeWave* Wave
- uint64_t alignment

8.3.8. struct ZIAdvisorHeader

```
typedef struct ZIAdvisorHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t reserved0[6];  
    uint8_t reserved1[8];  
} ZIAdvisorHeader;
```

Data Fields

- uint64_t sampleCount
Total sample count considered for advisor.
- uint8_t flags
Flags.
- uint8_t sampleFormat
Sample format Bode = 0, Step = 1, Impulse = 2.
- uint8_t reserved0
Reserved space for future use.
- uint8_t reserved1
Reserved space for future use.

8.3.9. struct ZIAdvisorSample

```
typedef struct ZIAdvisorSample {  
    double grid;  
    double x;  
    double y;  
} ZIAdvisorSample;
```

Data Fields

- double grid
Grid.
- double x
X.
- double y
Y.

8.3.10. struct ZIAdvisorWave

```
typedef struct ZIAdvisorWave {  
    ZITimeStamp timeStamp;  
    ZIAdvisorHeader header;  
    ZIAdvisorSample data[0];  
    union ZIAdvisorWave::@4 data;  
} ZIAdvisorWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZIAdvisorHeader header
- ZIAdvisorSample data
- union ZIAdvisorWave::@4 data
Advisor data vector.

8.3.11. struct ZIAsyncReply

```
typedef struct ZIAsyncReply {  
    ZITimeStamp timeStamp;  
    ZITimeStamp sampleTimeStamp;  
    uint16_t command;  
    uint16_t resultCode;  
    ZIAsyncTag tag;  
} ZIAsyncReply;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp of the reply (server clock)
- ZITimeStamp sampleTimeStamp
Time stamp of the target node sample, to which the reply belongs.
- uint16_t command
Command: 1 - ziAPIAsyncSetDoubleData 2 - ziAPIAsyncSetIntegerData 3 - ziAPIAsyncSetByteArray 4 - ziAPIAsyncSubscribe 5 - ziAPIAsyncUnSubscribe 6 - ziAPIAsyncGetValueAsPollData.
- uint16_t resultCode
Command result code (cast to ZIResult_enum)
- ZIAsyncTag tag
Tag sent along with the async command.

8.3.12. struct ZIAuxInSample

The structure used to hold data for a single auxiliary inputs sample.

```
#include "ziAPI.h"

typedef struct ZIAuxInSample {
    ZITimeStamp timeStamp;
    double ch0;
    double ch1;
} ZIAuxInSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- double ch0
Channel 0 voltage.
- double ch1
Channel 1 voltage.

8.3.13. struct ZIByteArray

The structure used to hold an arbitrary array of bytes. This is the version without timestamp used in API Level 1 compatibility mode.

```
#include "ziAPI.h"

typedef struct ZIByteArray {
    uint32_t length;
    uint8_t bytes[0];
} ZIByteArray;
```

Data Fields

- `uint32_t length`
Length of the data readable from the Bytes field.
- `uint8_t bytes`
The data itself. The array has the size given in length.

8.3.14. struct ZIByteArrayTS

The structure used to hold an arbitrary array of bytes. This is the same as [ZIByteArray](#), but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIByteArrayTS {
    ZITimeStamp timeStamp;
    uint32_t length;
    uint8_t bytes[0];
} ZIByteArrayTS;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint32_t length
length of the data readable from the bytes field
- uint8_t bytes
the data itself. The array has the size given in length

8.3.15. struct ZIChunkHeader

Structure to hold generic chunk data header information.

```
#include "ziAPI.h"

typedef struct ZIChunkHeader {
    ZITimeStamp systemTime;
    ZITimeStamp createdTimeStamp;
    ZITimeStamp changedTimeStamp;
    uint32_t flags;
    uint32_t moduleFlags;
    uint32_t status;
    uint32_t reserved0;
    uint64_t chunkSizeBytes;
    uint64_t triggerNumber;
    char name[32];
    uint32_t groupIndex;
    uint32_t color;
    uint32_t activeRow;
    uint32_t gridRows;
    uint32_t gridCols;
    uint32_t gridMode;
    uint32_t gridOperation;
    uint32_t gridDirection;
    uint32_t gridRepetitions;
    double gridColDelta;
    double gridColOffset;
    double gridRowDelta;
    double gridRowOffset;
    double bandwidth;
    double center;
    double nenbw;
} ZIChunkHeader;
```

Data Fields

- ZITimeStamp systemTime
System timestamp.
- ZITimeStamp createdTimeStamp
Creation timestamp.
- ZITimeStamp changedTimeStamp
Last changed timestamp.
- uint32_t flags
Flags (bitmask of values from ZIChunkHeaderFlags_enum)
- uint32_t moduleFlags
moduleFlags (bitmask of values from ZIChunkHeaderModuleFlags_enum, module-specific)
- uint32_t status
Status Flag: [0] : selected [1] : group assigned [2] : color edited [4] : name edited.
- uint32_t reserved0
- uint64_t chunkSizeBytes

Size in bytes used for memory usage calculation.

- `uint64_t triggerNumber`
SW Trigger counter since execution start.
- `char name`
Name in history list.
- `uint32_t groupIndex`
Group index in history list.
- `uint32_t color`
Color in history list.
- `uint32_t activeRow`
Active row in history list.
- `uint32_t gridRows`
Number of grid rows.
- `uint32_t gridCols`
Number of grid columns.
- `uint32_t gridMode`
Grid mode interpolation mode (0 = off, 1 = nearest, 2 = linear, 3 = Lanczos)
- `uint32_t gridOperation`
Grid mode operation (0 = replace, 1 = average)
- `uint32_t gridDirection`
Grid mode direction (0 = forward, 1 = revers, 2 = bidirectional)
- `uint32_t gridRepetitions`
Number of repetitions in grid mode.
- `double gridColDelta`
Delta between grid points in SI unit.
- `double gridColOffset`
Offset of first grid point relative to trigger.
- `double gridRowDelta`
Delta between grid rows in SI unit.
- `double gridRowOffset`
Delay of first grid row relative to trigger.
- `double bandwidth`
For FFT the bandwidth of the signal.
- `double center`
The FFT center frequency.
- `double nenbw`

For FFT the normalized effective noise bandwidth.

8.3.16. struct ZICntSample

The structure used to hold data for a single counter sample.

```
#include "ziAPI.h"

typedef struct ZICntSample {
    ZITimeStamp timeStamp;
    int32_t counter;
    uint32_t trigger;
} ZICntSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- int32_t counter
Counter value.
- uint32_t trigger
Trigger bits.

8.3.17. struct ZIComplexData

The structure used to hold a complex double value.

```
#include "ziAPI.h"

typedef struct ZIComplexData {
    ZITimeStamp timeStamp;
    ZIDoubleData real;
    ZIDoubleData imag;
} ZIComplexData;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the value has changed.
- ZIDoubleData real
- ZIDoubleData imag

8.3.18. struct ZIDemodSample

The structure used to hold data for a single demodulator sample.

```
#include "ziAPI.h"

typedef struct ZIDemodSample {
    ZITimeStamp timeStamp;
    double x;
    double y;
    double frequency;
    double phase;
    uint32_t dioBits;
    uint32_t trigger;
    double auxIn0;
    double auxIn1;
} ZIDemodSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the sample has been measured.
- double x
X part of the sample.
- double y
Y part of the sample.
- double frequency
oscillator frequency at that sample.
- double phase
oscillator phase at that sample.
- uint32_t dioBits
the current bits of the DIO.
- uint32_t trigger
trigger bits
- double auxIn0
value of Aux input 0.
- double auxIn1
value of Aux input 1.

8.3.19. struct ZIDIOSample

The structure used to hold data for a single digital I/O sample.

```
#include "ziAPI.h"

typedef struct ZIDIOSample {
    ZITimeStamp timeStamp;
    uint32_t bits;
    uint32_t reserved;
} ZIDIOSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- uint32_t bits
The digital I/O values.
- uint32_t reserved
Filler to keep 8 bytes alignment in the array of [ZIDIOSample](#) structures.

8.3.20. struct ZIDoubleDataTS

The structure used to hold a single IEEE double value. Same as ZIDoubleData, but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIDoubleDataTS {
    ZITimeStamp timeStamp;
    ZIDoubleData value;
} ZIDoubleDataTS;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the value has changed.
- ZIDoubleData value

8.3.21. struct ZIEvent

This struct holds event data forwarded by the Data Server.

```
#include "ziAPI.h"

typedef struct ZIEvent {
    uint32_t valueType;
    uint32_t count;
    uint8_t path[256];
    void* untyped;
    ZIDoubleData* doubleData;
    ZIDoubleDataTS* doubleDataTS;
    ZIIntegerData* integerData;
    ZIIntegerDataTS* integerDataTS;
    ZIComplexData* complexData;
    ZIByteArray* byteArray;
    ZIByteArrayTS* byteArrayTS;
    ZICntSample* cntSample;
    ZITrigSample* trigSample;
    ZITreeChangeData* treeChangeData;
    TreeChange* treeChangeDataOld;
    ZIDemodSample* demodSample;
    ZIAuxInSample* auxInSample;
    ZIDIOSample* dioSample;
    ZIScopeWave* scopeWave;
    ZIScopeWaveEx* scopeWaveEx;
    ScopeWave* scopeWaveOld;
    ZIPWAWave* pwaWave;
    ZISweeperWave* sweeperWave;
    ZISpectrumWave* spectrumWave;
    ZIAdvisorWave* advisorWave;
    ZIAsyncReply* asyncReply;
    ZIVectorData* vectorData;
    ZIImpedanceSample* impedanceSample;
    uint64_t alignment;
    union ZIEvent::@6 value;
    uint8_t data[0x400000];
} ZIEvent;
```

Data Fields

- `uint32_t valueType`
Specifies the type of the data held by the `ZIEvent`, see `ZIValueType_enum`.
- `uint32_t count`
Number of values available in this event.
- `uint8_t path`
The path to the node from which the event originates.
- `void* untyped`
For convenience. The void field doesn't have a corresponding data type.
- `ZIDoubleData* doubleData`
when `valueType == ZI_VALUE_TYPE_DOUBLE_DATA`
- `ZIDoubleDataTS* doubleDataTS`
when `valueType == ZI_VALUE_TYPE_DOUBLE_DATA_TS`

- `ZIntegerData*` `integerData`
when `valueType == ZI_VALUE_TYPE_INTEGER_DATA`
- `ZIntegerDataTS*` `integerDataTS`
when `valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS`
- `ZComplexData*` `complexData`
when `valueType == ZI_VALUE_TYPE_COMPLEX_DATA`
- `ZByteArray*` `byteArray`
when `valueType == ZI_VALUE_TYPE_BYTE_ARRAY`
- `ZByteArrayTS*` `byteArrayTS`
when `valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS`
- `ZCntSample*` `cntSample`
when `valueType == ZI_VALUE_TYPE_CNT_SAMPLE`
- `ZTrigSample*` `trigSample`
when `valueType == ZI_VALUE_TYPE_TRIG_SAMPLE`
- `ZTreeChangeData*` `treeChangeData`
when `valueType == ZI_VALUE_TYPE_TREE_CHANGE_DATA`
- `TreeChange*` `treeChangeDataOld`
when `valueType == ZI_VALUE_TYPE_TREE_CHANGE_DATA_OLD`
- `ZIDemodSample*` `demodSample`
when `valueType == ZI_VALUE_TYPE_DEMOD_SAMPLE`
- `ZAuxInSample*` `auxInSample`
when `valueType == ZI_VALUE_TYPE_AUXIN_SAMPLE`
- `ZIDIOSample*` `dioSample`
when `valueType == ZI_VALUE_TYPE_DIO_SAMPLE`
- `ZIScopeWave*` `scopeWave`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE`
- `ZIScopeWaveEx*` `scopeWaveEx`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE_EX`
- `ScopeWave*` `scopeWaveOld`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE_OLD`
- `ZIPWAWave*` `pwaWave`
when `valueType == ZI_VALUE_TYPE_PWA_WAVE`
- `ZISweeperWave*` `sweeperWave`
when `valueType == ZI_VALUE_TYPE_SWEEPER_WAVE`
- `ZISpectrumWave*` `spectrumWave`
when `valueType == ZI_VALUE_TYPE_SPECTRUM_WAVE`
- `ZIAdvisorWave*` `advisorWave`

- when valueType == ZI_VALUE_TYPE_ADVISOR_WAVE
- ZIAsyncReply* asyncReply
when valueType == ZI_VALUE_TYPE_ASYNC_REPLY
- ZIVectorData* vectorData
when valueType == ZI_VALUE_TYPE_VECTOR_DATA
- ZIImpedanceSample* impedanceSample
when valueType == ZI_VALUE_TYPE_IMPEDANCE_SAMPLE
- uint64_t alignment
ensure union size is 8 bytes
- union ZIEvent::@6 value
Convenience pointer to allow for access to the first entry in Data using the correct type according to ZIEvent.valueType field.
- uint8_t data
The raw value data.

Detailed Description

ZIEvent is used to give out events like value changes or errors to the user. Event handling functionality is provided by ziAPISubscribe and ziAPIUnSubscribe as well as ziAPIPollDataEx.

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZIEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", Event->value.doubleData[j]);

        break;

    case ZI_VALUE_TYPE_INTEGER_DATA:

        printf("%u elements of integer data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", (float)Event->value.integerData[j]);

        break;

    case ZI_VALUE_TYPE_DEMOD_SAMPLE:

        printf("%u elements of sample data %s\n",
```

```
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++)
        printf("TS=%f, X=%f, Y=%f.\n",
            (float)Event->value.demodSample[j].timeStamp,
            Event->value.demodSample[j].x,
            Event->value.demodSample[j].y);

    break;

case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

    printf("%u elements of tree-changed data, %s.\n",
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++) {
        switch (Event->value.treeChangeDataOld[j].Action) {
            case ZI_TREE_ACTION_REMOVE:
                printf("Tree removed: %s\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_ADD:
                printf("treeChangeDataOld added: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_CHANGE:
                printf("treeChangeDataOld changed: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;
        }
    }

    break;

default:

    printf("Unexpected event value type: %d.\n", Event->valueType);
    break;
}
```

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

8.3.22. struct ziEvent

This struct holds event data forwarded by the Data Server. Deprecated: See [ZiEvent](#).

```
#include "ziAPI.h"

typedef struct ziEvent {
    uint32_t Type;
    uint32_t Count;
    unsigned char Path[256];
    union ziEvent::Val Val;
    unsigned char Data[0x400000];
} ziEvent;
```

Data Structures

- union [ziEvent::Val](#)

Data Fields

- uint32_t Type
- uint32_t Count
- unsigned char Path
- union ziEvent::Val Val
- unsigned char Data

Detailed Description

[ziEvent](#) is used to give out events like value changes or errors to the user. Event handling functionality is provided by [ziAPISubscribe](#) and [ziAPIUnSubscribe](#) as well as [ziAPIPollDataEx](#).

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZiEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);
```

```
    for (j = 0; j < Event->count; j++)
        printf("%f\n", Event->value.doubleData[j]);

    break;

case ZI_VALUE_TYPE_INTEGER_DATA:

    printf("%u elements of integer data: %s.\n",
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++)
        printf("%f\n", (float)Event->value.integerData[j]);

    break;

case ZI_VALUE_TYPE_DEMOD_SAMPLE:

    printf("%u elements of sample data %s\n",
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++)
        printf("TS=%f, X=%f, Y=%f.\n",
            (float)Event->value.demodSample[j].timeStamp,
            Event->value.demodSample[j].x,
            Event->value.demodSample[j].y);

    break;

case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

    printf("%u elements of tree-changed data, %s.\n",
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++) {
        switch (Event->value.treeChangeDataOld[j].Action) {
            case ZI_TREE_ACTION_REMOVE:
                printf("Tree removed: %s\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_ADD:
                printf("treeChangeDataOld added: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_CHANGE:
                printf("treeChangeDataOld changed: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;
        }
    }

    break;

default:

    printf("Unexpected event value type: %d.\n", Event->valueType);
    break;
}
```

Data Structure Documentation

union ziEvent::Val

```
typedef union ziEvent::Val {  
    void* Void;  
    DemodSample* SampleDemod;  
    AuxInSample* SampleAuxIn;  
    DIOSample* SampleDIO;  
    ziDoubleType* Double;  
    ziIntegerType* Integer;  
    TreeChange* Tree;  
    ByteArrayData* ByteArray;  
    ScopeWave* Wave;  
    uint64_t alignment;  
} ziEvent::Val;
```

Data Fields

- void* Void
- [DemodSample*](#) SampleDemod
- [AuxInSample*](#) SampleAuxIn
- [DIOSample*](#) SampleDIO
- ziDoubleType* Double
- ziIntegerType* Integer
- [TreeChange*](#) Tree
- [ByteArrayData*](#) ByteArray
- [ScopeWave*](#) Wave
- uint64_t alignment

8.3.23. struct ZIImpedanceSample

The structure used to hold data for a single impedance sample.

```
#include "ziAPI.h"

typedef struct ZIImpedanceSample {
    ZITimeStamp timeStamp;
    double realz;
    double imagz;
    double frequency;
    double phase;
    uint32_t flags;
    uint32_t trigger;
    double param0;
    double param1;
    double drive;
    double bias;
} ZIImpedanceSample;
```

Data Fields

- ZITimeStamp timeStamp
Timestamp at which the sample has been measured.
- double realz
Real part of the impedance sample.
- double imagz
Imaginary part of the impedance sample.
- double frequency
Frequency at that sample.
- double phase
Phase at that sample.
- uint32_t flags
Flags (see [ZIImpFlags_enum](#))
- uint32_t trigger
Trigger bits.
- double param0
Value of model parameter 0.
- double param1
Value of model parameter 1.
- double drive
Drive amplitude.
- double bias
Bias voltage.

8.3.24. struct ZIIntegerDataTS

The structure used to hold a single 64bit signed integer value. Same as ZIIntegerData, but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIIntegerDataTS {
    ZITimeStamp timeStamp;
    ZIIntegerData value;
} ZIIntegerDataTS;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the value has changed.
- ZIIntegerData value

8.3.25. struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"

typedef struct ZIModuleEvent {
    uint64_t allocatedSize;
    ZIChunkHeader* header;
    ZIEvent
        value[0];
} ZIModuleEvent;
```

Data Fields

- `uint64_t allocatedSize`
For internal use - never modify!
- `ZIChunkHeader* header`
Chunk header.
- `ZIEvent value`
Defines location of stored `ZIEvent`.

8.3.26. struct ZIPWASample

Single PWA sample value.

```
#include "ziAPI.h"

typedef struct ZIPWASample {
    double binPhase;
    double x;
    double y;
    uint32_t countBin;
    uint32_t reserved;
} ZIPWASample;
```

Data Fields

- double binPhase
Phase position of each bin.
- double x
Real PWA result or X component of a demod PWA.
- double y
Y component of the demod PWA.
- uint32_t countBin
Number of events per bin.
- uint32_t reserved
Reserved.

8.3.27. struct ZIPWAWave

PWA Wave.

```
#include "ziAPI.h"

typedef struct ZIPWAWave {
    ZITimeStamp timeStamp;
    uint64_t sampleCount;
    uint32_t inputSelect;
    uint32_t oscSelect;
    uint32_t harmonic;
    uint32_t binCount;
    double frequency;
    uint8_t pwaType;
    uint8_t mode;
    uint8_t overflow;
    uint8_t commensurable;
    uint32_t reservedUInt;
    ZIPWASample
        data[0];
} ZIPWAWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint64_t sampleCount
Total sample count considered for PWA.
- uint32_t inputSelect
Input selection used for the PWA.
- uint32_t oscSelect
Oscillator used for the PWA.
- uint32_t harmonic
Harmonic setting.
- uint32_t binCount
Bin count of the PWA.
- double frequency
Frequency during PWA accumulation.
- uint8_t pwaType
Type of the PWA.
- uint8_t mode
PWA Mode [0: zoom PWA, 1: harmonic PWA].
- uint8_t overflow
Overflow indicators. overflow[0]: Data accumulator overflow, overflow[1]: Counter at limit, overflow[6..2]: Reserved, overflow[7]: Invalid (missing frames).
- uint8_t commensurable

Commensurability of the data.

- `uint32_t reservedUInt`
Reserved 32bit.
- `ZIPWASample` data
PWA data vector.

8.3.28. struct ZIScopeWave

The structure used to hold scope data (when using API Level 4). Note that `ZIScopeWave` does not contain the structure member `channelOffset`, whereas `ZIScopeWaveEx` does. The data may be formatted differently, depending on settings. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIScopeWave {
    ZITimeStamp timeStamp;
    ZITimeStamp triggerTimeStamp;
    double dt;
    uint8_t channelEnable[4];
    uint8_t channelInput[4];
    uint8_t triggerEnable;
    uint8_t triggerInput;
    uint8_t reserved0[2];
    uint8_t channelBWLimit[4];
    uint8_t channelMath[4];
    float channelScaling[4];
    uint32_t sequenceNumber;
    uint32_t segmentNumber;
    uint32_t blockNumber;
    uint64_t totalSamples;
    uint8_t dataTransferMode;
    uint8_t blockMarker;
    uint8_t flags;
    uint8_t sampleFormat;
    uint32_t sampleCount;
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    float dataFloat[0];
    union ZIScopeWave::@0 data;
} ZIScopeWave;
```

Data Fields

- `ZITimeStamp timeStamp`
The timestamp of the last sample in this data block.
- `ZITimeStamp triggerTimeStamp`
The timestamp of the trigger (may also fall between samples and in another block)
- `double dt`
Time difference between samples in seconds.
- `uint8_t channelEnable`
Up to four channels: if channel is enabled, corresponding element is non-zero.
- `uint8_t channelInput`
Specifies the input source for each of the scope four channels.

Value of `channelInput` and corresponding input source:

- 0 = Signal Input 1,
- 1 = Signal Input 2,

- 2 = Trigger Input 1,
- 3 = Trigger Input 2,
- 4 = Aux Output 1,
- 5 = Aux Output 2,
- 6 = Aux Output 3,
- 7 = Aux Output 4,
- 8 = Aux Input 1,
- 9 = Aux Input 2.
- uint8_t triggerEnable
Non-zero if trigger is enabled:

Bit encoded:
 - Bit (0): 1 = Trigger on rising edge,
 - Bit (1): 1 = Trigger on falling edge.
- uint8_t triggerInput
Trigger source (same values as for channel input)
- uint8_t reserved0
- uint8_t channelBWLimit
Bandwidth-limit flag, per channel.

Bit encoded:
 - Bit (0): 1 = Enable bandwidth limiting.
 - Bit (7...1): Reserved
- uint8_t channelMath
Enable/disable math operations such as averaging or FFT.

Bit encoded:
 - Bit(0): 1 = Perform averaging,
 - Bit(1): 1 = Perform FFT,
 - Bit(7...2): Reserved
- float channelScaling
Data scaling factors for up to 4 channels.
- uint32_t sequenceNumber
Current scope shot sequence number. Identifies a scope shot.
- uint32_t segmentNumber
Current segment number.
- uint32_t blockNumber
Current block number from the beginning of a scope shot.
Large scope shots are split into blocks, which need to be concatenated to obtain the complete scope shot.

- `uint64_t totalSamples`
Total number of samples in one channel in the current scope shot, same for all channels.
- `uint8_t dataTransferMode`
Data transfer mode.

Value and the corresponding data transfer mode:
 - 0 - SingleTransfer,
 - 1 - BlockTransfer,
 - 3 - ContinuousTransfer. Other values are reserved.
- `uint8_t blockMarker`
Block marker:

Bit encoded:
 - Bit (0): 1 = End marker for continuous or multi-block transfer,
 - Bit (7.0): Reserved.
- `uint8_t flags`
Indicator Flags.

Bit encoded:
 - Bit (0): 1 = Data loss detected (samples are 0),
 - Bit (1): 1 = Missed trigger,
 - Bit (2): 1 = Transfer failure (corrupted data).
- `uint8_t sampleFormat`
Data format of samples:

Value and the corresponding data format used:
 - 0 - Int16,
 - 1 - Int32,
 - 2 - Float,
 - 4 - Int16Interleaved,
 - 5 - Int32Interleaved,
 - 6 - FloatInterleaved.
- `uint32_t sampleCount`
Number of samples in one channel in the current block, same for all channels.
- `int16_t dataInt16`
Wave data when `sampleFormat==0` or `sampleFormat==4`.
- `int32_t dataInt32`
Wave data when `sampleFormat==1` or `sampleFormat==5`.
- `float dataFloat`
Wave data when `sampleFormat==2` or `sampleFormat==6`.

- union ZIScopeWave::@0 data

Wave data, access via union member dataInt16, dataInt32 or dataFloat depending on sampleFormat. Indexing scheme also depends on sampleFormat.

Example for interleaved int16 wave, 4096 samples, 2 channels:

- data.dataInt16[0] - sample 0 of channel 0,
- data.dataInt16[1] - sample 0 of channel 1,
- data.dataInt16[2] - sample 1 of channel 0,
- data.dataInt16[3] - sample 1 of channel 1,
- ...
- data.dataInt16[8190] - sample 4095 of channel 0,
- data.dataInt16[8191] - sample 4095 of channel 1.

Example for non-interleaved int16 wave, 4096 samples, 2 channels:

- data.dataInt16[0] - sample 0 of channel 0,
- data.dataInt16[1] - sample 1 of channel 0,
- .. - ...
- data.dataInt16[4095] - sample 4095 of channel 0,
- data.dataInt16[4096] - sample 0 of channel 1,
- data.dataInt16[4097] - sample 1 of channel 1,
- ...
- data.dataInt16[8191] - sample 4095 of channel 1.

8.3.29. struct ZIScopeWaveEx

The structure used to hold scope data (extended, when using API Level 5). Note that [ZIScopeWaveEx](#) contains the structure member `channelOffset`; [ZIScopeWave](#) does not. The data may be formatted differently, depending on settings. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIScopeWaveEx {
    ZITimeStamp timeStamp;
    ZITimeStamp triggerTimeStamp;
    double dt;
    uint8_t channelEnable[4];
    uint8_t channelInput[4];
    uint8_t triggerEnable;
    uint8_t triggerInput;
    uint8_t reserved0[2];
    uint8_t channelBWLimit[4];
    uint8_t channelMath[4];
    float channelScaling[4];
    uint32_t sequenceNumber;
    uint32_t segmentNumber;
    uint32_t blockNumber;
    uint64_t totalSamples;
    uint8_t dataTransferMode;
    uint8_t blockMarker;
    uint8_t flags;
    uint8_t sampleFormat;
    uint32_t sampleCount;
    double channelOffset[4];
    uint32_t totalSegments;
    uint32_t reserved1;
    uint64_t reserved2[31];
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    float dataFloat[0];
    union ZIScopeWaveEx::@1 data;
} ZIScopeWaveEx;
```

Data Fields

- `ZITimeStamp timeStamp`
The timestamp of the last sample in this data block.
- `ZITimeStamp triggerTimeStamp`
The Timestamp of the trigger (may also fall between samples and in another block).
- `double dt`
Time difference between samples in seconds.
- `uint8_t channelEnable`
Up to four channels: If channel is enabled, the corresponding element is non-zero.
- `uint8_t channelInput`
Specifies the input source for each of the scope four channels.

Value of `channelInput` and corresponding input source:

- 0 = Signal Input 1,
- 1 = Signal Input 2,
- 2 = Trigger Input 1,
- 3 = Trigger Input 2,
- 4 = Aux Output 1,
- 5 = Aux Output 2,
- 6 = Aux Output 3,
- 7 = Aux Output 4,
- 8 = Aux Input 1,
- 9 = Aux Input 2.
- `uint8_t triggerEnable`
Non-zero if trigger is enabled.

Bit encoded:
 - Bit (0): 1 = Trigger on rising edge,
 - Bit (1): 1 = Trigger on falling edge.
- `uint8_t triggerInput`
Trigger source (same values as for channel input)
- `uint8_t reserved0`
- `uint8_t channelBWLimit`
Bandwidth-limit flag, per channel.

Bit encoded:
 - Bit (0): 1 = Enable bandwidth limiting.
 - Bit (7..1): Reserved
- `uint8_t channelMath`
Enable/disable math operations such as averaging or FFT.

Bit encoded:
 - Bit(0): 1 = Perform averaging,
 - Bit(1): 1 = Perform FFT,
 - Bit(7..2): Reserved
- `float channelScaling`
Data scaling factors for up to 4 channels.
- `uint32_t sequenceNumber`
Current scope shot sequence number. Identifies a scope shot.
- `uint32_t segmentNumber`
Current segment number.
- `uint32_t blockNumber`

Current block number from the beginning of a scope shot.
Large scope shots are split into blocks, which need to be concatenated to obtain the complete scope shot.

- `uint64_t totalSamples`
Total number of samples in one channel in the current scope shot, same for all channels.

- `uint8_t dataTransferMode`
Data transfer mode.

Value and the corresponding data transfer mode:

- 0 - SingleTransfer,
- 1 - BlockTransfer,
- 3 - ContinuousTransfer. Other values are reserved.

- `uint8_t blockMarker`
Block marker providing additional information about the current block.

Bit encoded:

- Bit (0): 1 = End marker for continuous or multi-block transfer,
- Bit (7..0): Reserved.

- `uint8_t flags`
Indicator Flags.

Bit encoded:

- Bit (0): 1 = Data loss detected (samples are 0),
- Bit (1): 1 = Missed trigger,
- Bit (2): 1 = Transfer failure (corrupted data).
- Bit (3): 1 = Assembled scope recording. 'sampleCount' will be set to 0, use 'totalSamples' instead.
- Bit (7...4): Reserved.

- `uint8_t sampleFormat`
Data format of samples.

Value and the corresponding data format used:

- 0 - Int16,
- 1 - Int32,
- 2 - Float,
- 4 - Int16Interleaved,
- 5 - Int32Interleaved,
- 6 - FloatInterleaved.

- `uint32_t sampleCount`
Number of samples in one channel in the current block, same for all channels.

- `double channelOffset`
Data offset (scaled) for up to 4 channels.
- `uint32_t totalSegments`
Number of segments in the recording. Only valid if 'flags' bit (3) is set.
- `uint32_t reserved1`
- `uint64_t reserved2`
- `int16_t dataInt16`
Wave data when `sampleFormat==0` or `sampleFormat==4`.
- `int32_t dataInt32`
Wave data when `sampleFormat==1` or `sampleFormat==5`.
- `float dataFloat`
Wave data when `sampleFormat==2` or `sampleFormat==6`.
- `union ZIScopeWaveEx::@1 data`
Wave data, access via union member `dataInt16`, `dataInt32` or `dataFloat` depending on `sampleFormat`. Indexing scheme also depends on `sampleFormat`.

Example for interleaved int16 wave, 4096 samples, 2 channels:

- `data.dataInt16[0]` - sample 0 of channel 0,
- `data.dataInt16[1]` - sample 0 of channel 1,
- `data.dataInt16[2]` - sample 1 of channel 0,
- `data.dataInt16[3]` - sample 1 of channel 1,
- ...
- `data.dataInt16[8190]` - sample 4095 of channel 0,
- `data.dataInt16[8191]` - sample 4095 of channel 1.

Example for non-interleaved int16 wave, 4096 samples, 2 channels:

- `data.dataInt16[0]` - sample 0 of channel 0,
- `data.dataInt16[1]` - sample 1 of channel 0,
- .. - ...
- `data.dataInt16[4095]` - sample 4095 of channel 0,
- `data.dataInt16[4096]` - sample 0 of channel 1,
- `data.dataInt16[4097]` - sample 1 of channel 1,
- ...
- `data.dataInt16[8191]` - sample 4095 of channel 1.

8.3.30. struct ZISpectrumDemodSample

```
typedef struct ZISpectrumDemodSample {  
    double grid;  
    double filter;  
    double x;  
    double y;  
    double r;  
} ZISpectrumDemodSample;
```

Data Fields

- double grid
Grid.
- double filter
Filter strength at the specific grid point.
- double x
X.
- double y
Y.
- double r
R.

8.3.31. struct ZISpectrumHeader

```
typedef struct ZISpectrumHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t spectrumMode;  
    uint8_t window;  
    uint8_t reserved0[4];  
    uint8_t reserved1[8];  
    double bandwidth;  
    double rate;  
    double center;  
    double resolution;  
    double aliasingReject;  
    double nenbw;  
    double overlap;  
} ZISpectrumHeader;
```

Data Fields

- `uint64_t sampleCount`
Total sample count considered for spectrum.
- `uint8_t flags`
Flags Bit 0: Power Bit 1: Spectral density Bit 2: Absolute frequency Bit 3: Full span.
- `uint8_t sampleFormat`
Sample format Demodulator = 0.
- `uint8_t spectrumMode`
Spectrum mode FFT(x+iy) = 0, FFT(r) = 1, FFT(theta) = 2, FFT(freq) = 3, FFT(dtheta/dt)/2pi = 4.
- `uint8_t window`
Window Rectangular = 0, Hann = 1, Hamming = 2, Blackman Harris = 3, Exponential = 16 (ring-down), Cosine = 17 (ring-down), Cosine squared = 18 (ring-down)
- `uint8_t reserved0`
Reserved space for future use.
- `uint8_t reserved1`
Reserved space for future use.
- `double bandwidth`
Filter bandwidth.
- `double rate`
Rate of the sampled data.
- `double center`
FFT center value.
- `double resolution`
FFT bin resolution.

- `double aliasingReject`
Aliasing reject (dB)
- `double nenbw`
Correction factor for the used window when calculating spectral density.
- `double overlap`
FFT overlap [0 .. 1[.

8.3.32. struct ZISpectrumWave

```
typedef struct ZISpectrumWave {  
    ZITimeStamp timeStamp;  
    ZISpectrumHeader header;  
    ZISpectrumDemodSample dataDemod[0];  
    union ZISpectrumWave::@3 data;  
} ZISpectrumWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZISpectrumHeader header
- ZISpectrumDemodSample dataDemod
- union ZISpectrumWave::@3 data
Spectrum data vector.

8.3.33. struct ZIStatisticSample

```
typedef struct ZIStatisticSample {  
    double avg;  
    double stddev;  
    double pwr;  
} ZIStatisticSample;
```

Data Fields

- double avg
Average value or single value.
- double stddev
Standard deviation.
- double pwr
Power value.

8.3.34. struct ZISweeperDemodSample

```
typedef struct ZISweeperDemodSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    double tc;  
    double tcMeas;  
    double settling;  
    ZITimeStamp setTimeStamp;  
    ZITimeStamp nextTimeStamp;  
    ZIStatisticSample x;  
    ZIStatisticSample y;  
    ZIStatisticSample r;  
    ZIStatisticSample phase;  
    ZIStatisticSample frequency;  
    ZIStatisticSample auxin0;  
    ZIStatisticSample auxin1;  
} ZISweeperDemodSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Demodulator bandwidth used for the specific sweep point.
- uint64_t count
Sample count used for statistic calculation.
- double tc
Time constant calculated for the specific sweep point.
- double tcMeas
Time constant used by the device.
- double settling
Settling time (s) used to wait until averaging operation is started.
- ZITimeStamp setTimeStamp
Time stamp when the grid value was set on the device.
- ZITimeStamp nextTimeStamp
Time stamp when the first statistic value was recorded.
- ZIStatisticSample x
Sweep point statistic result of X.
- ZIStatisticSample y
Sweep point statistic result of Y.
- ZIStatisticSample r
Sweep point statistic result of R.
- ZIStatisticSample phase
Sweep point statistic result of phase.

- ZIStatisticSample frequency
Sweep point statistic result of frequency.
- ZIStatisticSample auxin0
Sweep point statistic result of auxin0.
- ZIStatisticSample auxin1
Sweep point statistic result of auxin1.

8.3.35. struct ZISweeperDoubleSample

```
typedef struct ZISweeperDoubleSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    ZIStatisticSample value;  
} ZISweeperDoubleSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Bandwidth.
- uint64_t count
Sample count used for statistic calculation.
- ZIStatisticSample value
Result value (y-axis)

8.3.36. struct ZISweeperHeader

```
typedef struct ZISweeperHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t sweepMode;  
    uint8_t bandwidthMode;  
    uint8_t reserved0[4];  
    uint8_t reserved1[8];  
} ZISweeperHeader;
```

Data Fields

- `uint64_t sampleCount`
Total sample count considered for sweeper.
- `uint8_t flags`
Flags Bit 0: Phase unwrap Bit 1: Sinc filter.
- `uint8_t sampleFormat`
Sample format Double = 0, Demodulator = 1, Impedance = 2.
- `uint8_t sweepMode`
Sweep mode Sequential = 0, Binary = 1, Bidirectional = 2, Reverse = 3.
- `uint8_t bandwidthMode`
Bandwidth mode Manual = 0, Fixed = 1, Auto = 2.
- `uint8_t reserved0`
Reserved space for future use.
- `uint8_t reserved1`
Reserved space for future use.

8.3.37. struct ZISweeperImpedanceSample

```
typedef struct ZISweeperImpedanceSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    double tc;  
    double tcMeas;  
    double settling;  
    ZITimeStamp setTimeStamp;  
    ZITimeStamp nextTimeStamp;  
    ZIStatisticSample realz;  
    ZIStatisticSample imagz;  
    ZIStatisticSample absz;  
    ZIStatisticSample phasez;  
    ZIStatisticSample frequency;  
    ZIStatisticSample param0;  
    ZIStatisticSample param1;  
    ZIStatisticSample drive;  
    ZIStatisticSample bias;  
} ZISweeperImpedanceSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Demodulator bandwidth used for the specific sweep point.
- uint64_t count
Sample count used for statistic calculation.
- double tc
Time constant calculated for the specific sweep point.
- double tcMeas
Time constant used by the device.
- double settling
Settling time (s) used to wait until averaging operation is started.
- ZITimeStamp setTimeStamp
Time stamp when the grid value was set on the device.
- ZITimeStamp nextTimeStamp
Time stamp when the first statistic value was recorded.
- ZIStatisticSample realz
Sweep point statistic result of X.
- ZIStatisticSample imagz
Sweep point statistic result of Y.
- ZIStatisticSample absz
Sweep point statistic result of R.
- ZIStatisticSample phasez

Sweep point statistic result of phase.

- ZIStatisticSample frequency
Sweep point statistic result of frequency.

- ZIStatisticSample param0
Sweep point statistic result of param0.

- ZIStatisticSample param1
Sweep point statistic result of param1.

- ZIStatisticSample drive
Sweep point statistic result of drive amplitude.

- ZIStatisticSample bias
Sweep point statistic result of bias.

8.3.38. struct ZISweeperWave

```
typedef struct ZISweeperWave {  
    ZITimeStamp timeStamp;  
    ZISweeperHeader header;  
    ZISweeperDoubleSample dataDouble[0];  
    ZISweeperDemodSample dataDemod[0];  
    ZISweeperImpedanceSample dataImpedance[0];  
    union ZISweeperWave::@2 data;  
} ZISweeperWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZISweeperHeader header
- ZISweeperDoubleSample dataDouble
- ZISweeperDemodSample dataDemod
- ZISweeperImpedanceSample dataImpedance
- union ZISweeperWave::@2 data
Sweeper data vector.

8.3.39. struct ZITreeChangeData

The struct is holding info about added or removed nodes.

```
#include "ziAPI.h"

typedef struct ZITreeChangeData {
    ZITimeStamp timeStamp;
    uint32_t action;
    char name[32];
} ZITreeChangeData;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint32_t action
field indicating which action occurred on the tree. A value of the ZITreeAction_enum.
- char name
Name of the Path that has been added, removed or changed.

8.3.40. struct ZITrigSample

The structure used to hold data for a single counter sample.

```
#include "ziAPI.h"

typedef struct ZITrigSample {
    ZITimeStamp timeStamp;
    ZITimeStamp sampleTick;
    uint32_t trigger;
    uint32_t missedTriggers;
    uint32_t awgTrigger;
    uint32_t dio;
    uint32_t sequenceIndex;
} ZITrigSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- ZITimeStamp sampleTick
The sample tick at which the values have been measured.
- uint32_t trigger
Trigger bits.
- uint32_t missedTriggers
Missed trigger bits.
- uint32_t awgTrigger
AWG trigger values at the time of trigger.
- uint32_t dio
Dio values at the time of trigger.
- uint32_t sequenceIndex
AWG sequencer index at the time of trigger.

8.3.41. struct ZIVectorData

The structure used to hold vector data block. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIVectorData {
    ZITimeStamp timeStamp;
    uint32_t sequenceNumber;
    uint32_t blockNumber;
    uint64_t totalElements;
    uint64_t blockOffset;
    uint32_t blockElements;
    uint8_t flags;
    uint8_t elementType;
    uint8_t reserved0[2];
    uint64_t reserved1[32];
    uint8_t dataUInt8[0];
    uint16_t dataUInt16[0];
    uint32_t dataUInt32[0];
    uint64_t dataUInt64[0];
    int8_t dataInt8[0];
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    int64_t dataInt64[0];
    double dataDouble[0];
    float dataFloat[0];
    union ZIVectorData::@5 data;
} ZIVectorData;
```

Data Fields

- **ZITimeStamp timeStamp**
Time stamp of this array data block.
- **uint32_t sequenceNumber**
Current array transfer sequence number. Incremented for each new transfer. Stays same for all blocks of a single array transfer.
- **uint32_t blockNumber**
Current block number from the beginning of an array transfer. Large array transfers are split into blocks, which need to be concatenated to obtain the complete array.
- **uint64_t totalElements**
Total number of elements in the array.
- **uint64_t blockOffset**
Offset of the current block first element from the beginning of the array.
- **uint32_t blockElements**
Number of elements in the current block.
- **uint8_t flags**
Block marker: Bit (0): 1 = End marker for multi-block transfer
Bit (1): 1 = Transfer failure Bit (7..2): Reserved.

- `uint8_t elementType`
Vector element type, see [ZIVectorElementType_enum](#).
- `uint8_t reserved0`
- `uint64_t reserved1`
- `uint8_t dataUInt8`
- `uint16_t dataUInt16`
- `uint32_t dataUInt32`
- `uint64_t dataUInt64`
- `int8_t dataInt8`
- `int16_t dataInt16`
- `int32_t dataInt32`
- `int64_t dataInt64`
- `double dataDouble`
- `float dataFloat`
- `union ZIVectorData::@5 data`
First data element of the current block.

8.4. File Documentation

8.4.1. File ziAPI.h

Header File for the LabOne C/C++ API.

```
#include "wchar.h"
```

Data Structures

- struct [ZIDoubleDataTS](#)
The structure used to hold a single IEEE double value. Same as ZIDoubleData, but with timestamp.
- struct [ZIIntegerDataTS](#)
The structure used to hold a single 64bit signed integer value. Same as ZIIntegerData, but with timestamp.
- struct [ZIDoubleComplexData](#)
The structure used to hold a complex double value.
- struct [ZITreeChangeData](#)
The struct is holding info about added or removed nodes.
- struct [TreeChange](#)
The structure used to hold info about added or removed nodes. This is the version without timestamp used in API v1 compatibility mode.
- struct [ZIDemodSample](#)
The structure used to hold data for a single demodulator sample.
- struct [ZIAuxInSample](#)
The structure used to hold data for a single auxiliary inputs sample.
- struct [ZIDIOSample](#)
The structure used to hold data for a single digital I/O sample.
- struct [ZIByteArray](#)
The structure used to hold an arbitrary array of bytes. This is the version without timestamp used in API Level 1 compatibility mode.
- struct [ZIByteArrayTS](#)
The structure used to hold an arbitrary array of bytes. This is the same as [ZIByteArray](#), but with timestamp.
- struct [ZICntSample](#)
The structure used to hold data for a single counter sample.
- struct [ZITrigSample](#)

The structure used to hold data for a single counter sample.

- struct [ScopeWave](#)

The structure used to hold a single scope shot (API Level 1). If the client is connected to the Data Server using API Level 4 (recommended if supported by your device class) please see [ZIScopeWave](#) instead ([ZIScopeWaveEx](#) for API Level 5 and above).

- struct [ZIScopeWave](#)

The structure used to hold scope data (when using API Level 4). Note that [ZIScopeWave](#) does not contain the structure member `channelOffset`, whereas [ZIScopeWaveEx](#) does. The data may be formatted differently, depending on settings. See the description of the structure members for details.

- struct [ZIScopeWaveEx](#)

The structure used to hold scope data (extended, when using API Level 5). Note that [ZIScopeWaveEx](#) contains the structure member `channelOffset`; [ZIScopeWave](#) does not. The data may be formatted differently, depending on settings. See the description of the structure members for details.

- struct [ZIPWASample](#)

Single PWA sample value.

- struct [ZIPWAWave](#)

PWA Wave.

- struct [ZIImpedanceSample](#)

The structure used to hold data for a single impedance sample.

- struct [ZISStatisticSample](#)

- struct [ZISweeperDoubleSample](#)

- struct [ZISweeperDemodSample](#)

- struct [ZISweeperImpedanceSample](#)

- struct [ZISweeperHeader](#)

- struct [ZISweeperWave](#)

- struct [ZISpectrumDemodSample](#)

- struct [ZISpectrumHeader](#)

- struct [ZISpectrumWave](#)
- struct [ZIAdvisorSample](#)
- struct [ZIAdvisorHeader](#)
- struct [ZIAdvisorWave](#)
- struct [ZIVectorData](#)
The structure used to hold vector data block. See the description of the structure members for details.
- struct [ZIAsyncReply](#)
- struct [ZIEvent](#)
This struct holds event data forwarded by the Data Server.
- struct [ZIChunkHeader](#)
Structure to hold generic chunk data header information.
- struct [ZIModuleEvent](#)
This struct holds data of a single chunk from module lookup.
- struct [DemodSample](#)
The [DemodSample](#) struct holds data for the `ZI_DATA_DEMODSAMPLE` data type. Deprecated: See [ZIDemodSample](#).
- struct [AuxInSample](#)
The [AuxInSample](#) struct holds data for the `ZI_DATA_AUXINSAMPLE` data type. Deprecated: See [ZIAuxInSample](#).
- struct [DIOSample](#)
The [DIOSample](#) struct holds data for the `ZI_DATA_DIOSAMPLE` data type. Deprecated: See [ZIDIOSample](#).
- struct [ByteArrayData](#)
The [ByteArrayData](#) struct holds data for the `ZI_DATA_BYTEARRAY` data type. Deprecated: See [ZIByteArray](#).
- struct [ziEvent](#)
This struct holds event data forwarded by the Data Server. Deprecated: See [ZIEvent](#).
- union [ziEvent::Val](#)

Defines

- `#define MAX_PATH_LEN 256`
The maximum length that has to be used for passing paths to functions (including terminating zero)
- `#define MAX_EVENT_SIZE 0x400000`
The maximum size of an event's data block.
- `#define MAX_NAME_LEN 32`
The maximum length of the node name (in tree change event)

Typedefs

- `typedef ZIModuleHandle`
A handle with which to reference an instance of a `ziCore` Module created with [ziAPIModCreate](#).
- `typedef ZIConnection`
The `ZIConnection` is a connection reference; it holds information and helper variables about a connection to the Data Server. There is nothing in this reference which the user may use, so it is hidden and instead a dummy pointer is used. See [ziAPIInit](#) for how to create a `ZIConnection`.
- `typedef ZIModuleEventPtr`
The pointer to a Module's data chunk to read out, updated via [ziAPIModGetChunk](#).

Enumerations

- `enum ZIResult_enum { ZI_INFO_BASE,`
`ZI_INFO_SUCCESS, ZI_INFO_MAX, ZI_WARNING_BASE,`
`ZI_WARNING_GENERAL, ZI_WARNING_UNDERRUN,`
`ZI_WARNING_OVERFLOW, ZI_WARNING_NOTFOUND,`
`ZI_WARNING_NO_ASYNC, ZI_WARNING_MAX,`
`ZI_ERROR_BASE, ZI_ERROR_GENERAL, ZI_ERROR_USB,`
`ZI_ERROR_MALLOC, ZI_ERROR_MUTEX_INIT,`
`ZI_ERROR_MUTEX_DESTROY, ZI_ERROR_MUTEX_LOCK,`
`ZI_ERROR_MUTEX_UNLOCK, ZI_ERROR_THREAD_START,`
`ZI_ERROR_THREAD_JOIN, ZI_ERROR_SOCKET_INIT,`
`ZI_ERROR_SOCKET_CONNECT, ZI_ERROR_HOSTNAME,`
`ZI_ERROR_CONNECTION, ZI_ERROR_TIMEOUT,`
`ZI_ERROR_COMMAND, ZI_ERROR_SERVER_INTERNAL,`
`ZI_ERROR_LENGTH, ZI_ERROR_FILE, ZI_ERROR_DUPLICATE,`
`ZI_ERROR_READONLY, ZI_ERROR_DEVICE_NOT_VISIBLE,`
`ZI_ERROR_DEVICE_IN_USE, ZI_ERROR_DEVICE_INTERFACE,`
`ZI_ERROR_DEVICE_CONNECTION_TIMEOUT,`
`ZI_ERROR_DEVICE_DIFFERENT_INTERFACE,`
`ZI_ERROR_DEVICE_NEEDS_FW_UPGRADE,`
`ZI_ERROR_ZIEVENT_DATATYPE_MISMATCH,`
`ZI_ERROR_DEVICE_NOT_FOUND,`


```
ZI_ERROR_NOT_SUPPORTED,  
ZI_ERROR_TOO_MANY_CONNECTIONS,  
ZI_ERROR_NOT_ON_HF2, ZI_ERROR_MAX }
```

Defines return value for all ziAPI functions. Divided into 3 regions: info, warning and error.

- enum [ZiValueType_enum](#) { ZI_VALUE_TYPE_NONE,
ZI_VALUE_TYPE_DOUBLE_DATA,
ZI_VALUE_TYPE_INTEGER_DATA,
ZI_VALUE_TYPE_DEMOD_SAMPLE,
ZI_VALUE_TYPE_SCOPE_WAVE_OLD,
ZI_VALUE_TYPE_AUXIN_SAMPLE,
ZI_VALUE_TYPE_DIO_SAMPLE,
ZI_VALUE_TYPE_BYTE_ARRAY, ZI_VALUE_TYPE_PWA_WAVE,
ZI_VALUE_TYPE_TREE_CHANGE_DATA_OLD,
ZI_VALUE_TYPE_DOUBLE_DATA_TS,
ZI_VALUE_TYPE_INTEGER_DATA_TS,
ZI_VALUE_TYPE_COMPLEX_DATA,
ZI_VALUE_TYPE_SCOPE_WAVE,
ZI_VALUE_TYPE_SCOPE_WAVE_EX,
ZI_VALUE_TYPE_BYTE_ARRAY_TS,
ZI_VALUE_TYPE_CNT_SAMPLE,
ZI_VALUE_TYPE_TRIG_SAMPLE,
ZI_VALUE_TYPE_TREE_CHANGE_DATA,
ZI_VALUE_TYPE_ASYNC_REPLY,
ZI_VALUE_TYPE_SWEEPER_WAVE,
ZI_VALUE_TYPE_SPECTRUM_WAVE,
ZI_VALUE_TYPE_ADVISOR_WAVE,
ZI_VALUE_TYPE_VECTOR_DATA,
ZI_VALUE_TYPE_IMPEDANCE_SAMPLE }

Enumerates all types that data in a [ZiEvent](#) may have.

- enum [ZiTreeAction_enum](#) { ZI_TREE_ACTION_REMOVE,
ZI_TREE_ACTION_ADD, ZI_TREE_ACTION_CHANGE }

Defines the actions that are performed on a tree,
as returned in the [ZiTreeChangeData::action](#) or
[ZiTreeChangeDataOld::action](#).

- enum [ZiImpFlags_enum](#) { ZI_IMP_FLAGS_NONE,
ZI_IMP_FLAGS_VALID_INTERNAL,
ZI_IMP_FLAGS_VALID_USER,
ZI_IMP_FLAGS_AUTORANGE_GATING,
ZI_IMP_FLAGS_OVERFLOW_VOLTAGE,
ZI_IMP_FLAGS_OVERFLOW_CURRENT,
ZI_IMP_FLAGS_UNDERFLOW_VOLTAGE,
ZI_IMP_FLAGS_UNDERFLOW_CURRENT,
ZI_IMP_FLAGS_FREQ_EXACT,
ZI_IMP_FLAGS_FREQ_INTERPOLATION,
ZI_IMP_FLAGS_FREQ_EXTRAPOLATION,
ZI_IMP_FLAGS_LOWDUT2T,
ZI_IMP_FLAGS_SUPPRESSION_PARAM0,
ZI_IMP_FLAGS_SUPPRESSION_PARAM1,
ZI_IMP_FLAGS_FREQLIMIT_RANGE_VOLTAGE,
ZI_IMP_FLAGS_FREQLIMIT_RANGE_CURRENT,
ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM0,
ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM1,
ZI_IMP_FLAGS_NEGATIVE_QFACTOR,

```
ZI_IMP_FLAGS_BWC_BIT0, ZI_IMP_FLAGS_BWC_BIT1,  
ZI_IMP_FLAGS_BWC_BIT2, ZI_IMP_FLAGS_BWC_BIT3,  
ZI_IMP_FLAGS_BWC_MASK,  
ZI_IMP_FLAGS_OPEN_DETECTION,  
ZI_IMP_FLAGS_OVERFLOW_SIGIN0,  
ZI_IMP_FLAGS_OVERFLOW_SIGIN1,  
ZI_IMP_FLAGS_MODEL_MASK }
```

Enumerates the bits set in an [ZIImpedanceSample](#)'s flags.

- [enum ZIVectorElementType_enum](#)
{ ZI_VECTOR_ELEMENT_TYPE_UINT8,
ZI_VECTOR_ELEMENT_TYPE_UINT16,
ZI_VECTOR_ELEMENT_TYPE_UINT32,
ZI_VECTOR_ELEMENT_TYPE_UINT64,
ZI_VECTOR_ELEMENT_TYPE_FLOAT,
ZI_VECTOR_ELEMENT_TYPE_DOUBLE,
ZI_VECTOR_ELEMENT_TYPE_ASCII }

Enumerates all the types that a [ZIVectorData::elementType](#) may have.

- [enum ZIAPIVersion_enum](#) { ZI_API_VERSION_0,
ZI_API_VERSION_1, ZI_API_VERSION_4, ZI_API_VERSION_5,
ZI_API_VERSION_6, ZI_API_VERSION_MAX }

- [enum ZIListNodes_enum](#) { ZI_LIST_NODES_NONE,
ZI_LIST_NODES_RECURSIVE, ZI_LIST_NODES_ABSOLUTE,
ZI_LIST_NODES_LEAFONLY,
ZI_LIST_NODES_SETTINGSONLY,
ZI_LIST_NODES_STREAMINGONLY,
ZI_LIST_NODES_SUBSCRIBEDONLY,
ZI_LIST_NODES_BASECHANNEL,
ZI_LIST_NODES_GETONLY }

Defines the values of the flags used in [ziAPIListNodes](#).

- [enum ZIChunkHeaderFlags_enum](#)
{ ZI_CHUNK_HEADER_FLAG_FINISHED,
ZI_CHUNK_HEADER_FLAG_ROLLMODE,
ZI_CHUNK_HEADER_FLAG_DATALOSS,
ZI_CHUNK_HEADER_FLAG_VALID,
ZI_CHUNK_HEADER_FLAG_DATA,
ZI_CHUNK_HEADER_FLAG_DISPLAY,
ZI_CHUNK_HEADER_FLAG_FREQDOMAIN,
ZI_CHUNK_HEADER_FLAG_SPECTRUM,
ZI_CHUNK_HEADER_FLAG_OVERLAPPED,
ZI_CHUNK_HEADER_FLAG_ROWFINISHED,
ZI_CHUNK_HEADER_FLAG_ONGRIDSAMPLING,
ZI_CHUNK_HEADER_FLAG_ROWREPETITION,
ZI_CHUNK_HEADER_FLAG_PREVIEW }

Defines the flags returned in the chunk header for all modules.

- [enum ZIChunkHeaderModuleFlags_enum](#)
{ ZI_CHUNK_HEADER_MODULE_FLAGS_WINDOW }

Defines flags returned in the chunk header that only apply for certain modules.

- enum [ZIVectorWriteStatus_enum](#)
{ ZI_VECTOR_WRITE_STATUS_IDLE,
ZI_VECTOR_WRITE_STATUS_PENDING }
- enum [TREE_ACTION](#) { TREE_ACTION_REMOVE,
TREE_ACTION_ADD, TREE_ACTION_CHANGE }
TREE_ACTION defines the values for the [TreeChange::Action](#) Variable.

Functions

- [ZIResult_enum](#) [ziAPIInit](#) ([ZIConnection*](#) conn)
Initializes a [ZIConnection](#) structure.
- [ZIResult_enum](#) [ziAPIDestroy](#) ([ZIConnection](#) conn)
Destroys a [ZIConnection](#) structure.
- [ZIResult_enum](#) [ziAPIConnect](#) ([ZIConnection](#) conn, const char* hostname, uint16_t port)
Connects the [ZIConnection](#) to Data Server.
- [ZIResult_enum](#) [ziAPIDisconnect](#) ([ZIConnection](#) conn)
Disconnects an established connection.
- [ZIResult_enum](#) [ziAPIListImplementations](#) (char* implementations, uint32_t bufferSize)
Returns the list of supported implementations.
- [ZIResult_enum](#) [ziAPIConnectEx](#) ([ZIConnection](#) conn, const char* hostname, uint16_t port, [ZIAPIVersion_enum](#) apiLevel, const char* implementation)
Connects to Data Server and enables extended [ziAPI](#).
- [ZIResult_enum](#) [ziAPIGetConnectionAPILevel](#) ([ZIConnection](#) conn, [ZIAPIVersion_enum*](#) apiLevel)
Returns [ziAPI](#) level used for the connection conn.
- [ZIResult_enum](#) [ziAPIGetVersion](#) (const char** version)
Retrieves the release version of [ziAPI](#).
- [ZIResult_enum](#) [ziAPIGetRevision](#) (unsigned int* revision)
Retrieves the revision of [ziAPI](#).
- [ZIResult_enum](#) [ziAPIListNodes](#) ([ZIConnection](#) conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)
Returns all child nodes found at the specified path.
- [ZIResult_enum](#) [ziAPIListNodesJSON](#) ([ZIConnection](#) conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)
Returns all child nodes found at the specified path.
- [ZIResult_enum](#) [ziAPIUpdateDevices](#) ([ZIConnection](#) conn)
Search for the newly connected devices and update the tree.

- `ZIResult_enum` `ziAPIConnectDevice ( ZIConnection conn, const char* deviceSerial, const char* deviceInterface, const char* interfaceParams )`
Connect a device to the server.
- `ZIResult_enum` `ziAPIDisconnectDevice ( ZIConnection conn, const char* deviceSerial )`
Disconnect a device from the server.
- `ZIResult_enum` `ziAPIGetValueD ( ZIConnection conn, const char* path, ZIDoubleData* value )`
gets the double-type value of the specified node
- `ZIResult_enum` `ziAPIGetComplexData ( ZIConnection conn, const char* path, ZIDoubleData* real, ZIDoubleData* imag )`
gets the complex double-type value of the specified node
- `ZIResult_enum` `ziAPIGetValueI ( ZIConnection conn, const char* path, ZIntegerData* value )`
gets the integer-type value of the specified node
- `ZIResult_enum` `ziAPIGetDemodSample ( ZIConnection conn, const char* path, ZIDemodSample* value )`
Gets the demodulator sample value of the specified node.
- `ZIResult_enum` `ziAPIGetDIOSample ( ZIConnection conn, const char* path, ZIDIOSample* value )`
Gets the Digital I/O sample of the specified node.
- `ZIResult_enum` `ziAPIGetAuxInSample ( ZIConnection conn, const char* path, ZIAuxInSample* value )`
gets the AuxIn sample of the specified node
- `ZIResult_enum` `ziAPIGetValueB ( ZIConnection conn, const char* path, unsigned char* buffer, unsigned int* length, unsigned int bufferSize )`
gets the Bytearray value of the specified node
- `ZIResult_enum` `ziAPIGetValueString ( ZIConnection conn, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize )`
gets a null-terminated string value of the specified node
- `ZIResult_enum` `ziAPIGetValueStringUnicode ( ZIConnection conn, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize )`
gets a null-terminated string value of the specified node
- `ZIResult_enum` `ziAPISetValueD ( ZIConnection conn, const char* path, ZIDoubleData value )`
asynchronously sets a double-type value to one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetComplexData ( ZIConnection conn, const char* path, ZIDoubleData real, ZIDoubleData imag )`

asynchronously sets a double-type complex value to one or more nodes specified in the path

- `ZIResult_enum` `ziAPISetValueI ( ZIConnection conn, const char* path, ZIIntegerData value )`
asynchronously sets an integer-type value to one or more nodes specified in a path
- `ZIResult_enum` `ziAPISetValueB ( ZIConnection conn, const char* path, unsigned char* buffer, unsigned int length )`
asynchronously sets the binary-type value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetValueString ( ZIConnection conn, const char* path, const char* str )`
asynchronously sets a string value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISetValueStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
asynchronously sets a unicode encoded string value of one or more nodes specified in the path
- `ZIResult_enum` `ziAPISyncSetValueD ( ZIConnection conn, const char* path, ZIDoubleData* value )`
synchronously sets a double-type value to one or more nodes specified in the path
- `ZIResult_enum` `ziAPISyncSetValueI ( ZIConnection conn, const char* path, ZIIntegerData* value )`
synchronously sets an integer-type value to one or more nodes specified in a path
- `ZIResult_enum` `ziAPISyncSetValueB ( ZIConnection conn, const char* path, uint8_t* buffer, uint32_t length, uint32_t bufferSize )`
Synchronously sets the binary-type value of one ore more nodes specified in the path.
- `ZIResult_enum` `ziAPISyncSetValueString ( ZIConnection conn, const char* path, const char* str )`
Synchronously sets a string value of one or more nodes specified in the path.
- `ZIResult_enum` `ziAPISyncSetValueStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
Synchronously sets a unicode string value of one or more nodes specified in the path.
- `ZIResult_enum` `ziAPISync ( ZIConnection conn )`
Synchronizes the session by dropping all pending data.
- `ZIResult_enum` `ziAPIEchoDevice ( ZIConnection conn, const char* deviceSerial )`
Sends an echo command to a device and blocks until answer is received.

- `ZIEvent* ziAPIAllocateEventEx ( )`
Allocates `ZIEvent` structure and returns the pointer to it.
Attention!!! It is the client code responsibility to deallocate the structure by calling `ziAPIDeallocateEventEx`!
- `void ziAPIDeallocateEventEx ( ZIEvent* ev )`
Deallocates `ZIEvent` structure created with `ziAPIAllocateEventEx()`.
- `ZIResult_enum ziAPISubscribe ( ZIConnection conn, const char* path )`
subscribes the nodes given by path for `ziAPIPollDataEx`
- `ZIResult_enum ziAPIUnSubscribe ( ZIConnection conn, const char* path )`
unsubscribes to the nodes given by path
- `ZIResult_enum ziAPIPollDataEx ( ZIConnection conn, ZIEvent* ev, uint32_t timeOutMilliseconds )`
checks if an event is available to read
- `ZIResult_enum ziAPIGetValueAsPollData ( ZIConnection conn, const char* path )`
triggers a value request, which will be given back on the poll event queue
- `ZIResult_enum ziAPIAsyncSetDoubleData ( ZIConnection conn, const char* path, ZIDoubleData value )`
- `ZIResult_enum ziAPIAsyncSetIntegerData ( ZIConnection conn, const char* path, ZIIntegerData value )`
- `ZIResult_enum ziAPIAsyncSetByteArray ( ZIConnection conn, const char* path, uint8_t* buffer, uint32_t length )`
- `ZIResult_enum ziAPIAsyncSetString ( ZIConnection conn, const char* path, const char* str )`
- `ZIResult_enum ziAPIAsyncSetStringUnicode ( ZIConnection conn, const char* path, const wchar_t* wstr )`
- `ZIResult_enum ziAPIAsyncSubscribe ( ZIConnection conn, const char* path, ZIAsyncTag tag )`
- `ZIResult_enum ziAPIAsyncUnSubscribe ( ZIConnection conn, const char* path, ZIAsyncTag tag )`
- `ZIResult_enum ziAPIAsyncGetValueAsPollData ( ZIConnection conn, const char* path, ZIAsyncTag tag )`

- `ZIResult_enum` `ziAPIGetError ( ZIResult_enum result, char** buffer, int* base )`
Returns a description and the severity for a `ZIResult_enum`.
- `ZIResult_enum` `ziAPIGetLastError ( ZIConnection conn, char* buffer, uint32_t bufferSize )`
Returns the message from the last error that occurred.
- `void` `ziAPISetDebugLevel ( int32_t debugLevel )`
Enable `ziAPI`'s log and set the severity level of entries to be included in the log.
- `void` `ziAPIWriteDebugLog ( int32_t debugLevel, const char* message )`
Write a message to `ziAPI`'s log with the specified severity.
- `ZIResult_enum` `ReadMEMFile ( const char* filename, char* buffer, int32_t bufferSize, int32_t* bytesUsed )`
- `ZIResult_enum` `ziAPIModCreate ( ZIConnection conn, ZIModuleHandle* handle, const char* moduleId )`
Create a `ZIModuleHandle` that can be used for asynchronous measurement tasks.
- `ZIResult_enum` `ziAPIModSetDoubleData ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIDoubleData value )`
Sets a module parameter to the specified double type.
- `ZIResult_enum` `ziAPIModSetIntegerData ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData value )`
Sets a module parameter to the specified integer type.
- `ZIResult_enum` `ziAPIModSetByteArray ( ZIConnection conn, ZIModuleHandle handle, const char* path, uint8_t* buffer, uint32_t length )`
Sets a module parameter to the specified byte array.
- `ZIResult_enum` `ziAPIModSetString ( ZIConnection conn, ZIModuleHandle handle, const char* path, const char* str )`
Sets a module parameter to the specified null-terminated string.
- `ZIResult_enum` `ziAPIModSetStringUnicode ( ZIConnection conn, ZIModuleHandle handle, const char* path, const wchar_t* wstr )`
Sets a module parameter to the specified null-terminated unicode string.
- `ZIResult_enum` `ziAPIModGetInteger ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData* value )`

- gets the integer-type value of the specified node
- `ZIResult_enum` `ziAPIModGetDouble ( ZIConnection conn, ZIModuleHandle handle, const char* path, ZIDoubleData* value )`
gets the double-type value of the specified node
 - `ZIResult_enum` `ziAPIModGetString ( ZIConnection conn, ZIModuleHandle handle, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize )`
gets the null-terminated string value of the specified node
 - `ZIResult_enum` `ziAPIModGetStringUnicode ( ZIConnection conn, ZIModuleHandle handle, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize )`
gets the null-terminated string value of the specified node
 - `ZIResult_enum` `ziAPIModListNodes ( ZIConnection conn, ZIModuleHandle handle, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags )`
Returns all child parameter node paths found under the specified parent module parameter path.
 - `ZIResult_enum` `ziAPIModSubscribe ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Subscribes to the nodes specified by path, these nodes will be recorded during module execution.
 - `ZIResult_enum` `ziAPIModUnSubscribe ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Unsubscribes to the nodes specified by path.
 - `ZIResult_enum` `ziAPIModExecute ( ZIConnection conn, ZIModuleHandle handle )`
Starts the module's thread and its associated measurement task.
 - `ZIResult_enum` `ziAPIModTrigger ( ZIConnection conn, ZIModuleHandle handle )`
Manually issue a trigger forcing data recording (SW Trigger Module only).
 - `ZIResult_enum` `ziAPIModProgress ( ZIConnection conn, ZIModuleHandle handle, ZIDoubleData* progress )`
Queries the current state of progress of the module's measurement task.
 - `ZIResult_enum` `ziAPIModFinished ( ZIConnection conn, ZIModuleHandle handle, ZIIntegerData* finished )`
Queries whether the module has finished its measurement task.
 - `ZIResult_enum` `ziAPIModFinish ( ZIConnection conn, ZIModuleHandle handle )`
Stops the module performing its measurement task.

- `ZIResult_enum` `ziAPIModSave ( ZIConnection conn, ZIModuleHandle handle, const char* fileName )`
Saves the currently accumulated data to file.
- `ZIResult_enum` `ziAPIModRead ( ZIConnection conn, ZIModuleHandle handle, const char* path )`
Make the currently accumulated data available for use in the C program.
- `ZIResult_enum` `ziAPIModNextNode ( ZIConnection conn, ZIModuleHandle handle, char* path, uint32_t bufferSize, ZIValueType_enum* valueType, uint64_t* chunks )`
Make the data for the next node available for reading with `ziAPIModGetChunk`.
- `ZIResult_enum` `ziAPIModGetChunk ( ZIConnection conn, ZIModuleHandle handle, uint64_t chunkIndex, ZIModuleEventPtr* ev )`
Get the specified data chunk from the current node.
- `ZIResult_enum` `ziAPIModEventDeallocate ( ZIConnection conn, ZIModuleHandle handle, ZIModuleEventPtr ev )`
Deallocate the `ZIModuleEventPtr` being used by the module.
- `ZIResult_enum` `ziAPIModClear ( ZIConnection conn, ZIModuleHandle handle )`
Terminates the module's thread and destroys the module.
- `ZIResult_enum` `ziAPIVectorWriteBlock ( ZIConnection conn, const char* path, ZIVectorData* vectorBlock )`
- `ZIResult_enum` `ziAPIVectorWriteGetStatus ( ZIConnection conn, const char* path, uint8_t* status )`
status - see `::ZIVectorWriteStatus_enum`
- `ZIResult_enum` `ziAPIVectorWrite ( ZIConnection conn, const char* path, const void* vectorPtr, uint8_t vectorElementType, uint64_t vectorSizeElements )`
vectorElementType - see `ZIVectorElementType_enum`
- `ZIResult_enum` `ziAPIDiscoveryFindAll ( ZIConnection conn, char* deviceId, uint32_t bufferSize )`
- `ZIResult_enum` `ziAPIDiscoveryFind ( ZIConnection conn, const char* deviceAddress, const char** deviceId )`
- `ZIResult_enum` `ziAPIDiscoveryGet ( ZIConnection conn, const char* deviceId, const char** propsJSON )`
- `ZIResult_enum` `ziAPIDiscoveryGetValue1 ( ZIConnection conn, const char* deviceId, const char* propName, ZIIntegerData* value )`

- `ZIResult_enum` `ziAPIDiscoveryGetValueS ( ZIConnection conn, const char* deviceId, const char* propName, const char** value )`
- `__inline ziEvent*` `ziAPIAllocateEvent ( )`
Deprecated: See `ziAPIAllocateEventEx()`.
- `__inline void` `ziAPIDeallocateEvent ( ziEvent* ev )`
Deprecated: See `ziAPIDeallocateEventEx()`.
- `__inline ZIResult_enum` `ziAPIPollData ( ZIConnection conn, ziEvent* ev, int timeOut )`
Checks if an event is available to read. Deprecated: See `ziAPIPollDataEx()`.
- `__inline ZIResult_enum` `ziAPIGetValueS ( ZIConnection conn, char* path, DemodSample* value )`
- `__inline ZIResult_enum` `ziAPIGetValueDIO ( ZIConnection conn, char* path, DIOSample* value )`
- `__inline ZIResult_enum` `ziAPIGetValueAuxIn ( ZIConnection conn, char* path, AuxInSample* value )`
- `double` `ziAPISecondsTimeStamp ( ziTimeStampType TS )`

Detailed Description

ziAPI provides all functionality to establish a connection with the Data Server and to communicate with it. It has functions for setting and getting parameters in a single call as well as an event-framework with which the user may subscribe the parameter tree and receive the events which occur when values change.

- All functions do not check passed pointers if they're NULL pointers. In that case a segmentation fault will occur.
- The ZIConnection is not thread-safe. One connection can only be used in one thread. If you want to use the ziAPI in a multi-threaded program you will have to use one ZIConnection for each thread that is communicating or implement a mutual exclusion.
- The Data Server is able to handle connections from threads simultaneously. The Data Server takes over the synchronization.

Data Structure Documentation

struct ZIDoubleDataTS

The structure used to hold a single IEEE double value. Same as ZIDoubleData, but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIDoubleDataTS {
    ZITimeStamp timeStamps;
    ZIDoubleData value;
} ZIDoubleDataTS;
```

Data Fields

- ZITimeStamp timeStamps
Time stamp at which the value has changed.
- ZIDoubleData value

struct ZIIntegerDataTS

The structure used to hold a single 64bit signed integer value. Same as ZIIntegerData, but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIIntegerDataTS {
    ZITimeStamp timeStamps;
    ZIIntegerData value;
} ZIIntegerDataTS;
```

Data Fields

- ZITimeStamp timeStamps
Time stamp at which the value has changed.
- ZIIntegerData value

struct ZIComplexData

The structure used to hold a complex double value.

```
#include "ziAPI.h"

typedef struct ZIComplexData {
    ZITimeStamp timeStamp;
    ZIDoubleData real;
    ZIDoubleData imag;
} ZIComplexData;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the value has changed.
- ZIDoubleData real
- ZIDoubleData imag

struct ZITreeChangeData

The struct is holding info about added or removed nodes.

```
#include "ziAPI.h"

typedef struct ZITreeChangeData {
    ZITimeStamp timeStamp;
    uint32_t action;
    char name[32];
} ZITreeChangeData;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint32_t action
field indicating which action occurred on the tree. A value of the ZITreeAction_enum.
- char name
Name of the Path that has been added, removed or changed.

struct TreeChange

The structure used to hold info about added or removed nodes. This is the version without timestamp used in API v1 compatibility mode.

```
#include "ziAPI.h"

typedef struct TreeChange {
    uint32_t Action;
    char Name[32];
} TreeChange;
```

Data Fields

- uint32_t Action
field indicating which action occurred on the tree. A value of the [ZITreeAction_enum](#) (TREE_ACTION) enum.
- char Name
Name of the Path that has been added, removed or changed.

struct ZIDemodSample

The structure used to hold data for a single demodulator sample.

```
#include "ziAPI.h"

typedef struct ZIDemodSample {
    ZITimeStamp timeStamp;
    double x;
    double y;
    double frequency;
    double phase;
    uint32_t dioBits;
    uint32_t trigger;
    double auxIn0;
    double auxIn1;
} ZIDemodSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the sample has been measured.
- double x
X part of the sample.
- double y
Y part of the sample.
- double frequency
oscillator frequency at that sample.
- double phase
oscillator phase at that sample.
- uint32_t dioBits
the current bits of the DIO.
- uint32_t trigger
trigger bits
- double auxIn0
value of Aux input 0.
- double auxIn1
value of Aux input 1.

struct ZIAuxInSample

The structure used to hold data for a single auxiliary inputs sample.

```
#include "ziAPI.h"

typedef struct ZIAuxInSample {
    ZITimeStamp timeStamp;
    double ch0;
    double ch1;
} ZIAuxInSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- double ch0
Channel 0 voltage.
- double ch1
Channel 1 voltage.

struct ZIDIOSample

The structure used to hold data for a single digital I/O sample.

```
#include "ziAPI.h"

typedef struct ZIDIOSample {
    ZITimeStamp timeStamp;
    uint32_t bits;
    uint32_t reserved;
} ZIDIOSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- uint32_t bits
The digital I/O values.
- uint32_t reserved
Filler to keep 8 bytes alignment in the array of [ZIDIOSample](#) structures.

struct ZIByteArray

The structure used to hold an arbitrary array of bytes. This is the version without timestamp used in API Level 1 compatibility mode.

```
#include "ziAPI.h"

typedef struct ZIByteArray {
    uint32_t length;
    uint8_t bytes[0];
} ZIByteArray;
```

Data Fields

- `uint32_t length`
Length of the data readable from the Bytes field.
- `uint8_t bytes`
The data itself. The array has the size given in length.

struct ZIByteArrayTS

The structure used to hold an arbitrary array of bytes. This is the same as [ZIByteArray](#), but with timestamp.

```
#include "ziAPI.h"

typedef struct ZIByteArrayTS {
    ZITimeStamp timeStamp;
    uint32_t length;
    uint8_t bytes[0];
} ZIByteArrayTS;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint32_t length
length of the data readable from the bytes field
- uint8_t bytes
the data itself. The array has the size given in length

struct ZICntSample

The structure used to hold data for a single counter sample.

```
#include "ziAPI.h"

typedef struct ZICntSample {
    ZITimeStamp timeStamp;
    int32_t counter;
    uint32_t trigger;
} ZICntSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- int32_t counter
Counter value.
- uint32_t trigger
Trigger bits.

struct ZITrigSample

The structure used to hold data for a single counter sample.

```
#include "ziAPI.h"

typedef struct ZITrigSample {
    ZITimeStamp timeStamp;
    ZITimeStamp sampleTick;
    uint32_t trigger;
    uint32_t missedTriggers;
    uint32_t awgTrigger;
    uint32_t dio;
    uint32_t sequenceIndex;
} ZITrigSample;
```

Data Fields

- ZITimeStamp timeStamp
The timestamp at which the values have been measured.
- ZITimeStamp sampleTick
The sample tick at which the values have been measured.
- uint32_t trigger
Trigger bits.
- uint32_t missedTriggers
Missed trigger bits.
- uint32_t awgTrigger
AWG trigger values at the time of trigger.
- uint32_t dio
Dio values at the time of trigger.
- uint32_t sequenceIndex
AWG sequencer index at the time of trigger.

struct ScopeWave

The structure used to hold a single scope shot (API Level 1). If the client is connected to the Data Server using API Level 4 (recommended if supported by your device class) please see [ZIScopeWave](#) instead ([ZIScopeWaveEx](#) for API Level 5 and above).

```
#include "ziAPI.h"

typedef struct ScopeWave {
    double dt;
    uint32_t ScopeChannel;
    uint32_t TriggerChannel;
    uint32_t BWLimit;
    uint32_t Count;
    int16_t Data[0];
} ScopeWave;
```

Data Fields

- double dt
Time difference between samples.
- uint32_t ScopeChannel
Scope channel of the represented data.
- uint32_t TriggerChannel
Trigger channel of the represented data.
- uint32_t BWLimit
Bandwidth-limit flag.
- uint32_t Count
Count of samples.
- int16_t Data
First wave data.

struct ZIScopeWave

The structure used to hold scope data (when using API Level 4). Note that `ZIScopeWave` does not contain the structure member `channelOffset`, whereas `ZIScopeWaveEx` does. The data may be formatted differently, depending on settings. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIScopeWave {
    ZITimeStamp timeStamp;
    ZITimeStamp triggerTimeStamp;
    double dt;
    uint8_t channelEnable[4];
    uint8_t channelInput[4];
    uint8_t triggerEnable;
    uint8_t triggerInput;
    uint8_t reserved0[2];
    uint8_t channelBWLimit[4];
    uint8_t channelMath[4];
    float channelScaling[4];
    uint32_t sequenceNumber;
    uint32_t segmentNumber;
    uint32_t blockNumber;
    uint64_t totalSamples;
    uint8_t dataTransferMode;
    uint8_t blockMarker;
    uint8_t flags;
    uint8_t sampleFormat;
    uint32_t sampleCount;
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    float dataFloat[0];
    union ZIScopeWave::@0 data;
} ZIScopeWave;
```

Data Fields

- `ZITimeStamp timeStamp`
The timestamp of the last sample in this data block.
- `ZITimeStamp triggerTimeStamp`
The timestamp of the trigger (may also fall between samples and in another block)
- `double dt`
Time difference between samples in seconds.
- `uint8_t channelEnable`
Up to four channels: if channel is enabled, corresponding element is non-zero.
- `uint8_t channelInput`
Specifies the input source for each of the scope four channels.

Value of `channelInput` and corresponding input source:

- 0 = Signal Input 1,
- 1 = Signal Input 2,
- 2 = Trigger Input 1,

- 3 = Trigger Input 2,
- 4 = Aux Output 1,
- 5 = Aux Output 2,
- 6 = Aux Output 3,
- 7 = Aux Output 4,
- 8 = Aux Input 1,
- 9 = Aux Input 2.
- `uint8_t triggerEnable`
Non-zero if trigger is enabled:

Bit encoded:
 - Bit (0): 1 = Trigger on rising edge,
 - Bit (1): 1 = Trigger on falling edge.
- `uint8_t triggerInput`
Trigger source (same values as for channel input)
- `uint8_t reserved0`
- `uint8_t channelBWLimit`
Bandwidth-limit flag, per channel.

Bit encoded:
 - Bit (0): 1 = Enable bandwidth limiting.
 - Bit (7...1): Reserved
- `uint8_t channelMath`
Enable/disable math operations such as averaging or FFT.

Bit encoded:
 - Bit(0): 1 = Perform averaging,
 - Bit(1): 1 = Perform FFT,
 - Bit(7...2): Reserved
- `float channelScaling`
Data scaling factors for up to 4 channels.
- `uint32_t sequenceNumber`
Current scope shot sequence number. Identifies a scope shot.
- `uint32_t segmentNumber`
Current segment number.
- `uint32_t blockNumber`
Current block number from the beginning of a scope shot.
Large scope shots are split into blocks, which need to be concatenated to obtain the complete scope shot.
- `uint64_t totalSamples`

Total number of samples in one channel in the current scope shot, same for all channels.

- `uint8_t dataTransferMode`

Data transfer mode.

Value and the corresponding data transfer mode:

- 0 - SingleTransfer,
- 1 - BlockTransfer,
- 3 - ContinuousTransfer. Other values are reserved.

- `uint8_t blockMarker`

Block marker:

Bit encoded:

- Bit (0): 1 = End marker for continuous or multi-block transfer,
- Bit (7..0): Reserved.

- `uint8_t flags`

Indicator Flags.

Bit encoded:

- Bit (0): 1 = Data loss detected (samples are 0),
- Bit (1): 1 = Missed trigger,
- Bit (2): 1 = Transfer failure (corrupted data).

- `uint8_t sampleFormat`

Data format of samples:

Value and the corresponding data format used:

- 0 - Int16,
- 1 - Int32,
- 2 - Float,
- 4 - Int16Interleaved,
- 5 - Int32Interleaved,
- 6 - FloatInterleaved.

- `uint32_t sampleCount`

Number of samples in one channel in the current block, same for all channels.

- `int16_t dataInt16`

Wave data when `sampleFormat==0` or `sampleFormat==4`.

- `int32_t dataInt32`

Wave data when `sampleFormat==1` or `sampleFormat==5`.

- `float dataFloat`

Wave data when `sampleFormat==2` or `sampleFormat==6`.

- union ZIScopeWave::@0 data

Wave data, access via union member dataInt16, dataInt32 or dataFloat depending on sampleFormat. Indexing scheme also depends on sampleFormat.

Example for interleaved int16 wave, 4096 samples, 2 channels:

- data.dataInt16[0] - sample 0 of channel 0,
- data.dataInt16[1] - sample 0 of channel 1,
- data.dataInt16[2] - sample 1 of channel 0,
- data.dataInt16[3] - sample 1 of channel 1,
- ...
- data.dataInt16[8190] - sample 4095 of channel 0,
- data.dataInt16[8191] - sample 4095 of channel 1.

Example for non-interleaved int16 wave, 4096 samples, 2 channels:

- data.dataInt16[0] - sample 0 of channel 0,
- data.dataInt16[1] - sample 1 of channel 0,
- .. - ...
- data.dataInt16[4095] - sample 4095 of channel 0,
- data.dataInt16[4096] - sample 0 of channel 1,
- data.dataInt16[4097] - sample 1 of channel 1,
- ...
- data.dataInt16[8191] - sample 4095 of channel 1.

struct ZIScopeWaveEx

The structure used to hold scope data (extended, when using API Level 5). Note that [ZIScopeWaveEx](#) contains the structure member `channelOffset`; [ZIScopeWave](#) does not. The data may be formatted differently, depending on settings. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIScopeWaveEx {
    ZITimeStamp timeStamp;
    ZITimeStamp triggerTimeStamp;
    double dt;
    uint8_t channelEnable[4];
    uint8_t channelInput[4];
    uint8_t triggerEnable;
    uint8_t triggerInput;
    uint8_t reserved0[2];
    uint8_t channelBWLimit[4];
    uint8_t channelMath[4];
    float channelScaling[4];
    uint32_t sequenceNumber;
    uint32_t segmentNumber;
    uint32_t blockNumber;
    uint64_t totalSamples;
    uint8_t dataTransferMode;
    uint8_t blockMarker;
    uint8_t flags;
    uint8_t sampleFormat;
    uint32_t sampleCount;
    double channelOffset[4];
    uint32_t totalSegments;
    uint32_t reserved1;
    uint64_t reserved2[31];
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    float dataFloat[0];
    union ZIScopeWaveEx::@1 data;
} ZIScopeWaveEx;
```

Data Fields

- `ZITimeStamp timeStamp`
The timestamp of the last sample in this data block.
- `ZITimeStamp triggerTimeStamp`
The Timestamp of the trigger (may also fall between samples and in another block).
- `double dt`
Time difference between samples in seconds.
- `uint8_t channelEnable`
Up to four channels: If channel is enabled, the corresponding element is non-zero.
- `uint8_t channelInput`
Specifies the input source for each of the scope four channels.

Value of `channelInput` and corresponding input source:

- 0 = Signal Input 1,
- 1 = Signal Input 2,
- 2 = Trigger Input 1,
- 3 = Trigger Input 2,
- 4 = Aux Output 1,
- 5 = Aux Output 2,
- 6 = Aux Output 3,
- 7 = Aux Output 4,
- 8 = Aux Input 1,
- 9 = Aux Input 2.
- `uint8_t triggerEnable`
Non-zero if trigger is enabled.

Bit encoded:
 - Bit (0): 1 = Trigger on rising edge,
 - Bit (1): 1 = Trigger on falling edge.
- `uint8_t triggerInput`
Trigger source (same values as for channel input)
- `uint8_t reserved0`
- `uint8_t channelBWLimit`
Bandwidth-limit flag, per channel.

Bit encoded:
 - Bit (0): 1 = Enable bandwidth limiting.
 - Bit (7...1): Reserved
- `uint8_t channelMath`
Enable/disable math operations such as averaging or FFT.

Bit encoded:
 - Bit(0): 1 = Perform averaging,
 - Bit(1): 1 = Perform FFT,
 - Bit(7...2): Reserved
- `float channelScaling`
Data scaling factors for up to 4 channels.
- `uint32_t sequenceNumber`
Current scope shot sequence number. Identifies a scope shot.
- `uint32_t segmentNumber`
Current segment number.
- `uint32_t blockNumber`

Current block number from the beginning of a scope shot.
Large scope shots are split into blocks, which need to be concatenated to obtain the complete scope shot.

- `uint64_t totalSamples`
Total number of samples in one channel in the current scope shot, same for all channels.

- `uint8_t dataTransferMode`
Data transfer mode.

Value and the corresponding data transfer mode:

- 0 - SingleTransfer,
- 1 - BlockTransfer,
- 3 - ContinuousTransfer. Other values are reserved.

- `uint8_t blockMarker`
Block marker providing additional information about the current block.

Bit encoded:

- Bit (0): 1 = End marker for continuous or multi-block transfer,
- Bit (7..0): Reserved.

- `uint8_t flags`
Indicator Flags.

Bit encoded:

- Bit (0): 1 = Data loss detected (samples are 0),
- Bit (1): 1 = Missed trigger,
- Bit (2): 1 = Transfer failure (corrupted data).
- Bit (3): 1 = Assembled scope recording. 'sampleCount' will be set to 0, use 'totalSamples' instead.
- Bit (7...4): Reserved.

- `uint8_t sampleFormat`
Data format of samples.

Value and the corresponding data format used:

- 0 - Int16,
- 1 - Int32,
- 2 - Float,
- 4 - Int16Interleaved,
- 5 - Int32Interleaved,
- 6 - FloatInterleaved.

- `uint32_t sampleCount`
Number of samples in one channel in the current block, same for all channels.

- `double channelOffset`
Data offset (scaled) for up to 4 channels.
- `uint32_t totalSegments`
Number of segments in the recording. Only valid if 'flags' bit (3) is set.
- `uint32_t reserved1`
- `uint64_t reserved2`
- `int16_t dataInt16`
Wave data when `sampleFormat==0` or `sampleFormat==4`.
- `int32_t dataInt32`
Wave data when `sampleFormat==1` or `sampleFormat==5`.
- `float dataFloat`
Wave data when `sampleFormat==2` or `sampleFormat==6`.
- `union ZIScopeWaveEx::@1 data`
Wave data, access via union member `dataInt16`, `dataInt32` or `dataFloat` depending on `sampleFormat`. Indexing scheme also depends on `sampleFormat`.

Example for interleaved int16 wave, 4096 samples, 2 channels:

- `data.dataInt16[0]` - sample 0 of channel 0,
- `data.dataInt16[1]` - sample 0 of channel 1,
- `data.dataInt16[2]` - sample 1 of channel 0,
- `data.dataInt16[3]` - sample 1 of channel 1,
- ...
- `data.dataInt16[8190]` - sample 4095 of channel 0,
- `data.dataInt16[8191]` - sample 4095 of channel 1.

Example for non-interleaved int16 wave, 4096 samples, 2 channels:

- `data.dataInt16[0]` - sample 0 of channel 0,
- `data.dataInt16[1]` - sample 1 of channel 0,
- .. - ...
- `data.dataInt16[4095]` - sample 4095 of channel 0,
- `data.dataInt16[4096]` - sample 0 of channel 1,
- `data.dataInt16[4097]` - sample 1 of channel 1,
- ...
- `data.dataInt16[8191]` - sample 4095 of channel 1.

struct ZIPWASample

Single PWA sample value.

```
#include "ziAPI.h"

typedef struct ZIPWASample {
    double binPhase;
    double x;
    double y;
    uint32_t countBin;
    uint32_t reserved;
} ZIPWASample;
```

Data Fields

- double binPhase
Phase position of each bin.
- double x
Real PWA result or X component of a demod PWA.
- double y
Y component of the demod PWA.
- uint32_t countBin
Number of events per bin.
- uint32_t reserved
Reserved.

struct ZIPWAWave

PWA Wave.

```
#include "ziAPI.h"

typedef struct ZIPWAWave {
    ZITimeStamp timeStamp;
    uint64_t sampleCount;
    uint32_t inputSelect;
    uint32_t oscSelect;
    uint32_t harmonic;
    uint32_t binCount;
    double frequency;
    uint8_t pwaType;
    uint8_t mode;
    uint8_t overflow;
    uint8_t commensurable;
    uint32_t reservedUInt;
    ZIPWASample
        data[0];
} ZIPWAWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- uint64_t sampleCount
Total sample count considered for PWA.
- uint32_t inputSelect
Input selection used for the PWA.
- uint32_t oscSelect
Oscillator used for the PWA.
- uint32_t harmonic
Harmonic setting.
- uint32_t binCount
Bin count of the PWA.
- double frequency
Frequency during PWA accumulation.
- uint8_t pwaType
Type of the PWA.
- uint8_t mode
PWA Mode [0: zoom PWA, 1: harmonic PWA].
- uint8_t overflow
Overflow indicators. overflow[0]: Data accumulator overflow, overflow[1]: Counter at limit, overflow[6..2]: Reserved, overflow[7]: Invalid (missing frames).
- uint8_t commensurable
Commensurability of the data.

- `uint32_t reservedUInt`
Reserved 32bit.
- `ZIPWASample` data
PWA data vector.

struct ZIImpedanceSample

The structure used to hold data for a single impedance sample.

```
#include "ziAPI.h"

typedef struct ZIImpedanceSample {
    ZITimeStamp timeStamp;
    double realz;
    double imagz;
    double frequency;
    double phase;
    uint32_t flags;
    uint32_t trigger;
    double param0;
    double param1;
    double drive;
    double bias;
} ZIImpedanceSample;
```

Data Fields

- ZITimeStamp timeStamp
Timestamp at which the sample has been measured.
- double realz
Real part of the impedance sample.
- double imagz
Imaginary part of the impedance sample.
- double frequency
Frequency at that sample.
- double phase
Phase at that sample.
- uint32_t flags
Flags (see [ZIImpFlags_enum](#))
- uint32_t trigger
Trigger bits.
- double param0
Value of model parameter 0.
- double param1
Value of model parameter 1.
- double drive
Drive amplitude.
- double bias
Bias voltage.

struct ZIStatisticSample

```
typedef struct ZIStatisticSample {  
    double avg;  
    double stddev;  
    double pwr;  
} ZIStatisticSample;
```

Data Fields

- double avg
Average value or single value.
- double stddev
Standard deviation.
- double pwr
Power value.

struct ZISweeperDoubleSample

```
typedef struct ZISweeperDoubleSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    ZIStatisticSample value;  
} ZISweeperDoubleSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Bandwidth.
- uint64_t count
Sample count used for statistic calculation.
- ZIStatisticSample value
Result value (y-axis)

struct ZISweeperDemodSample

```
typedef struct ZISweeperDemodSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    double tc;  
    double tcMeas;  
    double settling;  
    ZITimeStamp setTimeStamp;  
    ZITimeStamp nextTimeStamp;  
    ZIStatisticSample x;  
    ZIStatisticSample y;  
    ZIStatisticSample r;  
    ZIStatisticSample phase;  
    ZIStatisticSample frequency;  
    ZIStatisticSample auxin0;  
    ZIStatisticSample auxin1;  
} ZISweeperDemodSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Demodulator bandwidth used for the specific sweep point.
- uint64_t count
Sample count used for statistic calculation.
- double tc
Time constant calculated for the specific sweep point.
- double tcMeas
Time constant used by the device.
- double settling
Settling time (s) used to wait until averaging operation is started.
- ZITimeStamp setTimeStamp
Time stamp when the grid value was set on the device.
- ZITimeStamp nextTimeStamp
Time stamp when the first statistic value was recorded.
- ZIStatisticSample x
Sweep point statistic result of X.
- ZIStatisticSample y
Sweep point statistic result of Y.
- ZIStatisticSample r
Sweep point statistic result of R.
- ZIStatisticSample phase
Sweep point statistic result of phase.

- ZIStatisticSample frequency
Sweep point statistic result of frequency.
- ZIStatisticSample auxin0
Sweep point statistic result of auxin0.
- ZIStatisticSample auxin1
Sweep point statistic result of auxin1.

struct ZISweeperImpedanceSample

```
typedef struct ZISweeperImpedanceSample {  
    double grid;  
    double bandwidth;  
    uint64_t count;  
    double tc;  
    double tcMeas;  
    double settling;  
    ZITimeStamp setTimeStamp;  
    ZITimeStamp nextTimeStamp;  
    ZIStatisticSample realz;  
    ZIStatisticSample imagz;  
    ZIStatisticSample absz;  
    ZIStatisticSample phasez;  
    ZIStatisticSample frequency;  
    ZIStatisticSample param0;  
    ZIStatisticSample param1;  
    ZIStatisticSample drive;  
    ZIStatisticSample bias;  
} ZISweeperImpedanceSample;
```

Data Fields

- double grid
Grid value (x-axis)
- double bandwidth
Demodulator bandwidth used for the specific sweep point.
- uint64_t count
Sample count used for statistic calculation.
- double tc
Time constant calculated for the specific sweep point.
- double tcMeas
Time constant used by the device.
- double settling
Settling time (s) used to wait until averaging operation is started.
- ZITimeStamp setTimeStamp
Time stamp when the grid value was set on the device.
- ZITimeStamp nextTimeStamp
Time stamp when the first statistic value was recorded.
- ZIStatisticSample realz
Sweep point statistic result of X.
- ZIStatisticSample imagz
Sweep point statistic result of Y.
- ZIStatisticSample absz
Sweep point statistic result of R.
- ZIStatisticSample phasez

Sweep point statistic result of phase.

- ZIStatisticSample frequency
Sweep point statistic result of frequency.

- ZIStatisticSample param0
Sweep point statistic result of param0.

- ZIStatisticSample param1
Sweep point statistic result of param1.

- ZIStatisticSample drive
Sweep point statistic result of drive amplitude.

- ZIStatisticSample bias
Sweep point statistic result of bias.

struct ZISweeperHeader

```
typedef struct ZISweeperHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t sweepMode;  
    uint8_t bandwidthMode;  
    uint8_t reserved0[4];  
    uint8_t reserved1[8];  
} ZISweeperHeader;
```

Data Fields

- `uint64_t sampleCount`
Total sample count considered for sweeper.
- `uint8_t flags`
Flags Bit 0: Phase unwrap Bit 1: Sinc filter.
- `uint8_t sampleFormat`
Sample format Double = 0, Demodulator = 1, Impedance = 2.
- `uint8_t sweepMode`
Sweep mode Sequential = 0, Binary = 1, Bidirectional = 2, Reverse = 3.
- `uint8_t bandwidthMode`
Bandwidth mode Manual = 0, Fixed = 1, Auto = 2.
- `uint8_t reserved0`
Reserved space for future use.
- `uint8_t reserved1`
Reserved space for future use.

struct ZISweeperWave

```
typedef struct ZISweeperWave {  
    ZITimeStamp timeStamp;  
    ZISweeperHeader header;  
    ZISweeperDoubleSample dataDouble[0];  
    ZISweeperDemodSample dataDemod[0];  
    ZISweeperImpedanceSample dataImpedance[0];  
    union ZISweeperWave::@2 data;  
} ZISweeperWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZISweeperHeader header
- ZISweeperDoubleSample dataDouble
- ZISweeperDemodSample dataDemod
- ZISweeperImpedanceSample dataImpedance
- union ZISweeperWave::@2 data
Sweeper data vector.

struct ZISpectrumDemodSample

```
typedef struct ZISpectrumDemodSample {  
    double grid;  
    double filter;  
    double x;  
    double y;  
    double r;  
} ZISpectrumDemodSample;
```

Data Fields

- double grid
Grid.
- double filter
Filter strength at the specific grid point.
- double x
X.
- double y
Y.
- double r
R.

struct ZISpectrumHeader

```
typedef struct ZISpectrumHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t spectrumMode;  
    uint8_t window;  
    uint8_t reserved0[4];  
    uint8_t reserved1[8];  
    double bandwidth;  
    double rate;  
    double center;  
    double resolution;  
    double aliasingReject;  
    double nenbw;  
    double overlap;  
} ZISpectrumHeader;
```

Data Fields

- `uint64_t sampleCount`
Total sample count considered for spectrum.
- `uint8_t flags`
Flags Bit 0: Power Bit 1: Spectral density Bit 2: Absolute frequency Bit 3: Full span.
- `uint8_t sampleFormat`
Sample format Demodulator = 0.
- `uint8_t spectrumMode`
Spectrum mode FFT(x+iy) = 0, FFT(r) = 1, FFT(theta) = 2, FFT(freq) = 3, FFT(dtheta/dt)/2pi = 4.
- `uint8_t window`
Window Rectangular = 0, Hann = 1, Hamming = 2, Blackman Harris = 3, Exponential = 16 (ring-down), Cosine = 17 (ring-down), Cosine squared = 18 (ring-down)
- `uint8_t reserved0`
Reserved space for future use.
- `uint8_t reserved1`
Reserved space for future use.
- `double bandwidth`
Filter bandwidth.
- `double rate`
Rate of the sampled data.
- `double center`
FFT center value.
- `double resolution`
FFT bin resolution.
- `double aliasingReject`

Aliasing reject (dB)

- double nenbw
Correction factor for the used window when calculating spectral density.
- double overlap
FFT overlap [0 .. 1[.

struct ZISpectrumWave

```
typedef struct ZISpectrumWave {  
    ZITimeStamp timeStamp;  
    ZISpectrumHeader header;  
    ZISpectrumDemodSample dataDemod[0];  
    union ZISpectrumWave::@3 data;  
} ZISpectrumWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZISpectrumHeader header
- ZISpectrumDemodSample dataDemod
- union ZISpectrumWave::@3 data
Spectrum data vector.

struct ZIAdvisorSample

```
typedef struct ZIAdvisorSample {  
    double grid;  
    double x;  
    double y;  
} ZIAdvisorSample;
```

Data Fields

- double grid
Grid.
- double x
X.
- double y
Y.

struct ZIAdvisorHeader

```
typedef struct ZIAdvisorHeader {  
    uint64_t sampleCount;  
    uint8_t flags;  
    uint8_t sampleFormat;  
    uint8_t reserved0[6];  
    uint8_t reserved1[8];  
} ZIAdvisorHeader;
```

Data Fields

- `uint64_t sampleCount`
Total sample count considered for advisor.
- `uint8_t flags`
Flags.
- `uint8_t sampleFormat`
Sample format Bode = 0, Step = 1, Impulse = 2.
- `uint8_t reserved0`
Reserved space for future use.
- `uint8_t reserved1`
Reserved space for future use.

struct ZIAdvisorWave

```
typedef struct ZIAdvisorWave {  
    ZITimeStamp timeStamp;  
    ZIAdvisorHeader header;  
    ZIAdvisorSample data[0];  
    union ZIAdvisorWave::@4 data;  
} ZIAdvisorWave;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp at which the data was updated.
- ZIAdvisorHeader header
- ZIAdvisorSample data
- union ZIAdvisorWave::@4 data
Advisor data vector.

struct ZIVectorData

The structure used to hold vector data block. See the description of the structure members for details.

```
#include "ziAPI.h"

typedef struct ZIVectorData {
    ZITimeStamp timeStamp;
    uint32_t sequenceNumber;
    uint32_t blockNumber;
    uint64_t totalElements;
    uint64_t blockOffset;
    uint32_t blockElements;
    uint8_t flags;
    uint8_t elementType;
    uint8_t reserved0[2];
    uint64_t reserved1[32];
    uint8_t dataUInt8[0];
    uint16_t dataUInt16[0];
    uint32_t dataUInt32[0];
    uint64_t dataUInt64[0];
    int8_t dataInt8[0];
    int16_t dataInt16[0];
    int32_t dataInt32[0];
    int64_t dataInt64[0];
    double dataDouble[0];
    float dataFloat[0];
    union ZIVectorData::@5 data;
} ZIVectorData;
```

Data Fields

- **ZITimeStamp timeStamp**
Time stamp of this array data block.
- **uint32_t sequenceNumber**
Current array transfer sequence number. Incremented for each new transfer. Stays same for all blocks of a single array transfer.
- **uint32_t blockNumber**
Current block number from the beginning of an array transfer. Large array transfers are split into blocks, which need to be concatenated to obtain the complete array.
- **uint64_t totalElements**
Total number of elements in the array.
- **uint64_t blockOffset**
Offset of the current block first element from the beginning of the array.
- **uint32_t blockElements**
Number of elements in the current block.
- **uint8_t flags**
Block marker: Bit (0): 1 = End marker for multi-block transfer
Bit (1): 1 = Transfer failure Bit (7..2): Reserved.
- **uint8_t elementType**

Vector element type, see [ZIVectorElementType_enum](#).

- uint8_t reserved0
- uint64_t reserved1
- uint8_t dataUInt8
- uint16_t dataUInt16
- uint32_t dataUInt32
- uint64_t dataUInt64
- int8_t dataInt8
- int16_t dataInt16
- int32_t dataInt32
- int64_t dataInt64
- double dataDouble
- float dataFloat
- union ZIVectorData::@5 data
First data element of the current block.

struct ZIAsyncReply

```
typedef struct ZIAsyncReply {  
    ZITimeStamp timeStamp;  
    ZITimeStamp sampleTimeStamp;  
    uint16_t command;  
    uint16_t resultCode;  
    ZIAsyncTag tag;  
} ZIAsyncReply;
```

Data Fields

- ZITimeStamp timeStamp
Time stamp of the reply (server clock)
- ZITimeStamp sampleTimeStamp
Time stamp of the target node sample, to which the reply belongs.
- uint16_t command
Command: 1 - ziAPIAsyncSetDoubleData 2 - ziAPIAsyncSetIntegerData 3 - ziAPIAsyncSetByteArray 4 - ziAPIAsyncSubscribe 5 - ziAPIAsyncUnSubscribe 6 - ziAPIAsyncGetValueAsPollData.
- uint16_t resultCode
Command result code (cast to ZIResult_enum)
- ZIAsyncTag tag
Tag sent along with the async command.

struct ZIEvent

This struct holds event data forwarded by the Data Server.

```
#include "ziAPI.h"

typedef struct ZIEvent {
    uint32_t valueType;
    uint32_t count;
    uint8_t path[256];
    void* untyped;
    ZIDoubleData* doubleData;
    ZIDoubleDataTS* doubleDataTS;
    ZIIIntegerData* integerData;
    ZIIIntegerDataTS* integerDataTS;
    ZIComplexData* complexData;
    ZIByteArray* byteArray;
    ZIByteArrayTS* byteArrayTS;
    ZICntSample* cntSample;
    ZITrigSample* trigSample;
    ZITreeChangeData* treeChangeData;
    TreeChange* treeChangeDataOld;
    ZIDemodSample* demodSample;
    ZIAuxInSample* auxInSample;
    ZIDIOSample* dioSample;
    ZIScopeWave* scopeWave;
    ZIScopeWaveEx* scopeWaveEx;
    ScopeWave* scopeWaveOld;
    ZIPWAWave* pwaWave;
    ZISweeperWave* sweeperWave;
    ZISpectrumWave* spectrumWave;
    ZIAdvisorWave* advisorWave;
    ZIAsyncReply* asyncReply;
    ZIVectorData* vectorData;
    ZIImpedanceSample* impedanceSample;
    uint64_t alignment;
    union ZIEvent::@6 value;
    uint8_t data[0x400000];
} ZIEvent;
```

Data Fields

- `uint32_t valueType`
Specifies the type of the data held by the [ZIEvent](#), see [ZIValueType_enum](#).
- `uint32_t count`
Number of values available in this event.
- `uint8_t path`
The path to the node from which the event originates.
- `void* untyped`
For convenience. The void field doesn't have a corresponding data type.
- `ZIDoubleData* doubleData`
when `valueType == ZI_VALUE_TYPE_DOUBLE_DATA`
- `ZIDoubleDataTS* doubleDataTS`
when `valueType == ZI_VALUE_TYPE_DOUBLE_DATA_TS`

- `ZIntegerData* integerData`
when `valueType == ZI_VALUE_TYPE_INTEGER_DATA`
- `ZIntegerDataTS* integerDataTS`
when `valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS`
- `ZComplexData* complexData`
when `valueType == ZI_VALUE_TYPE_COMPLEX_DATA`
- `ZByteArray* byteArray`
when `valueType == ZI_VALUE_TYPE_BYTE_ARRAY`
- `ZByteArrayTS* byteArrayTS`
when `valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS`
- `ZCntSample* cntSample`
when `valueType == ZI_VALUE_TYPE_CNT_SAMPLE`
- `ZTrigSample* trigSample`
when `valueType == ZI_VALUE_TYPE_TRIG_SAMPLE`
- `ZTreeChangeData* treeChangeData`
when `valueType == ZI_VALUE_TYPE_TREE_CHANGE_DATA`
- `TreeChange* treeChangeDataOld`
when `valueType == ZI_VALUE_TYPE_TREE_CHANGE_DATA_OLD`
- `ZIDemodSample* demodSample`
when `valueType == ZI_VALUE_TYPE_DEMOD_SAMPLE`
- `ZAuxInSample* auxInSample`
when `valueType == ZI_VALUE_TYPE_AUXIN_SAMPLE`
- `ZIDIOSample* dioSample`
when `valueType == ZI_VALUE_TYPE_DIO_SAMPLE`
- `ZIScopeWave* scopeWave`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE`
- `ZIScopeWaveEx* scopeWaveEx`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE_EX`
- `ScopeWave* scopeWaveOld`
when `valueType == ZI_VALUE_TYPE_SCOPE_WAVE_OLD`
- `ZIPWAWave* pwaWave`
when `valueType == ZI_VALUE_TYPE_PWA_WAVE`
- `ZISweeperWave* sweeperWave`
when `valueType == ZI_VALUE_TYPE_SWEEPER_WAVE`
- `ZISpectrumWave* spectrumWave`
when `valueType == ZI_VALUE_TYPE_SPECTRUM_WAVE`
- `ZIAdvisorWave* advisorWave`

- when valueType == ZI_VALUE_TYPE_ADVISOR_WAVE
- ZIAsyncReply* asyncReply
when valueType == ZI_VALUE_TYPE_ASYNC_REPLY
- ZIVectorData* vectorData
when valueType == ZI_VALUE_TYPE_VECTOR_DATA
- ZIImpedanceSample* impedanceSample
when valueType == ZI_VALUE_TYPE_IMPEDANCE_SAMPLE
- uint64_t alignment
ensure union size is 8 bytes
- union ZIEvent::@6 value
Convenience pointer to allow for access to the first entry in Data using the correct type according to [ZIEvent.valueType](#) field.
- uint8_t data
The raw value data.

Detailed Description

[ZIEvent](#) is used to give out events like value changes or errors to the user. Event handling functionality is provided by [ziAPISubscribe](#) and [ziAPIUnSubscribe](#) as well as [ziAPIPollDataEx](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZIEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", Event->value.doubleData[j]);

        break;

    case ZI_VALUE_TYPE_INTEGER_DATA:

        printf("%u elements of integer data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", (float)Event->value.integerData[j]);

        break;

    case ZI_VALUE_TYPE_DEMOD_SAMPLE:

        printf("%u elements of sample data %s\n",
            Event->count,
```



```
        Event->path);

    for (j = 0; j < Event->count; j++)
        printf("TS=%f, X=%f, Y=%f.\n",
            (float)Event->value.demodSample[j].timeStamp,
            Event->value.demodSample[j].x,
            Event->value.demodSample[j].y);

    break;

case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

    printf("%u elements of tree-changed data, %s.\n",
        Event->count,
        Event->path);

    for (j = 0; j < Event->count; j++) {
        switch (Event->value.treeChangeDataOld[j].Action) {
            case ZI_TREE_ACTION_REMOVE:
                printf("Tree removed: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_ADD:
                printf("treeChangeDataOld added: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;

            case ZI_TREE_ACTION_CHANGE:
                printf("treeChangeDataOld changed: %s.\n",
                    Event->value.treeChangeDataOld[j].Name);
                break;
        }
    }

    break;

default:

    printf("Unexpected event value type: %d.\n", Event->valueType);
    break;
}
}
```

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

struct ZIChunkHeader

Structure to hold generic chunk data header information.

```
#include "ziAPI.h"

typedef struct ZIChunkHeader {
    ZITimeStamp systemTime;
    ZITimeStamp createdTimeStamp;
    ZITimeStamp changedTimeStamp;
    uint32_t flags;
    uint32_t moduleFlags;
    uint32_t status;
    uint32_t reserved0;
    uint64_t chunkSizeBytes;
    uint64_t triggerNumber;
    char name[32];
    uint32_t groupIndex;
    uint32_t color;
    uint32_t activeRow;
    uint32_t gridRows;
    uint32_t gridCols;
    uint32_t gridMode;
    uint32_t gridOperation;
    uint32_t gridDirection;
    uint32_t gridRepetitions;
    double gridColDelta;
    double gridColOffset;
    double gridRowDelta;
    double gridRowOffset;
    double bandwidth;
    double center;
    double nenbw;
} ZIChunkHeader;
```

Data Fields

- ZITimeStamp systemTime
System timestamp.
- ZITimeStamp createdTimeStamp
Creation timestamp.
- ZITimeStamp changedTimeStamp
Last changed timestamp.
- uint32_t flags
Flags (bitmask of values from ZIChunkHeaderFlags_enum)
- uint32_t moduleFlags
moduleFlags (bitmask of values from ZIChunkHeaderModuleFlags_enum, module-specific)
- uint32_t status
Status Flag: [0] : selected [1] : group assigned [2] : color edited [4] : name edited.
- uint32_t reserved0
- uint64_t chunkSizeBytes

Size in bytes used for memory usage calculation.

- `uint64_t triggerNumber`
SW Trigger counter since execution start.
- `char name`
Name in history list.
- `uint32_t groupIndex`
Group index in history list.
- `uint32_t color`
Color in history list.
- `uint32_t activeRow`
Active row in history list.
- `uint32_t gridRows`
Number of grid rows.
- `uint32_t gridCols`
Number of grid columns.
- `uint32_t gridMode`
Grid mode interpolation mode (0 = off, 1 = nearest, 2 = linear, 3 = Lanczos)
- `uint32_t gridOperation`
Grid mode operation (0 = replace, 1 = average)
- `uint32_t gridDirection`
Grid mode direction (0 = forward, 1 = revers, 2 = bidirectional)
- `uint32_t gridRepetitions`
Number of repetitions in grid mode.
- `double gridColDelta`
Delta between grid points in SI unit.
- `double gridColOffset`
Offset of first grid point relative to trigger.
- `double gridRowDelta`
Delta between grid rows in SI unit.
- `double gridRowOffset`
Delay of first grid row relative to trigger.
- `double bandwidth`
For FFT the bandwidth of the signal.
- `double center`
The FFT center frequency.
- `double nenbw`

For FFT the normalized effective noise bandwidth.

struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"

typedef struct ZIModuleEvent {
    uint64_t allocatedSize;
    ZIChunkHeader* header;
    ZIEvent
        value[0];
} ZIModuleEvent;
```

Data Fields

- `uint64_t allocatedSize`
For internal use - never modify!
- `ZIChunkHeader* header`
Chunk header.
- `ZIEvent value`
Defines location of stored `ZIEvent`.

struct DemodSample

The [DemodSample](#) struct holds data for the ZI_DATA_DEMODSAMPLE data type. Deprecated: See [ZIDemodSample](#).

```
#include "ziAPI.h"

typedef struct DemodSample {
    ziTimeStampType TimeStamp;
    double X;
    double Y;
    double Frequency;
    double Phase;
    unsigned int DIOBits;
    unsigned int Reserved;
    double AuxIn0;
    double AuxIn1;
} DemodSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `double X`
- `double Y`
- `double Frequency`
- `double Phase`
- `unsigned int DIOBits`
- `unsigned int Reserved`
- `double AuxIn0`
- `double AuxIn1`

struct AuxInSample

The [AuxInSample](#) struct holds data for the ZI_DATA_AUXINSAMPLE data type. Deprecated: See [ZIAuxInSample](#).

```
#include "ziAPI.h"

typedef struct AuxInSample {
    ziTimeStampType TimeStamp;
    double Ch0;
    double Ch1;
} AuxInSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `double Ch0`
- `double Ch1`

struct DIOSample

The [DIOSample](#) struct holds data for the ZI_DATA_DIOSAMPLE data type. Deprecated: See [ZIDIOSample](#).

```
#include "ziAPI.h"

typedef struct DIOSample {
    ziTimeStampType TimeStamp;
    unsigned int Bits;
    unsigned int Reserved;
} DIOSample;
```

Data Fields

- `ziTimeStampType TimeStamp`
- `unsigned int Bits`
- `unsigned int Reserved`

struct ByteArrayData

The [ByteArrayData](#) struct holds data for the ZI_DATA_BYTEARRAY data type. Deprecated: See [ZIByteArray](#).

```
#include "ziAPI.h"

typedef struct ByteArrayData {
    unsigned int Len;
    unsigned char Bytes[0];
} ByteArrayData;
```

Data Fields

- unsigned int Len
- unsigned char Bytes

struct `ziEvent`

This struct holds event data forwarded by the Data Server. Deprecated: See [ZIEvent](#).

```
#include "ziAPI.h"

typedef struct ziEvent {
    uint32_t Type;
    uint32_t Count;
    unsigned char Path[256];
    union ziEvent::Val Val;
    unsigned char Data[0x400000];
} ziEvent;
```

Data Structures

- union `ziEvent::Val`

Data Fields

- `uint32_t` Type
- `uint32_t` Count
- unsigned char Path
- union `ziEvent::Val` Val
- unsigned char Data

Detailed Description

[ziEvent](#) is used to give out events like value changes or errors to the user. Event handling functionality is provided by `ziAPISubscribe` and `ziAPIUnSubscribe` as well as `ziAPIPollDataEx`.

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

void ProcessEvent(ZIEvent* Event) {
    unsigned int j;

    switch (Event->valueType) {
    case ZI_VALUE_TYPE_DOUBLE_DATA:

        printf("%u elements of double data: %s.\n",
            Event->count,
            Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", Event->value.doubleData[j]);
    }
```

```
        break;

    case ZI_VALUE_TYPE_INTEGER_DATA:

        printf("%u elements of integer data: %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++)
            printf("%f\n", (float)Event->value.integerData[j]);

        break;

    case ZI_VALUE_TYPE_DEMOD_SAMPLE:

        printf("%u elements of sample data %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++)
            printf("TS=%f, X=%f, Y=%f.\n",
                   (float)Event->value.demodSample[j].timeStamp,
                   Event->value.demodSample[j].x,
                   Event->value.demodSample[j].y);

        break;

    case ZI_VALUE_TYPE_TREE_CHANGE_DATA:

        printf("%u elements of tree-changed data, %s.\n",
               Event->count,
               Event->path);

        for (j = 0; j < Event->count; j++) {
            switch (Event->value.treeChangeDataOld[j].Action) {
                case ZI_TREE_ACTION_REMOVE:
                    printf("Tree removed: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;

                case ZI_TREE_ACTION_ADD:
                    printf("treeChangeDataOld added: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;

                case ZI_TREE_ACTION_CHANGE:
                    printf("treeChangeDataOld changed: %s.\n",
                           Event->value.treeChangeDataOld[j].Name);
                    break;
            }
        }

        break;

    default:

        printf("Unexpected event value type: %d.\n", Event->valueType);
        break;
    }
}
```

Data Structure Documentation

union ziEvent::Val

```
typedef union ziEvent::Val {  
    void* Void;  
    DemodSample* SampleDemod;  
    AuxInSample* SampleAuxIn;  
    DIOSample* SampleDIO;  
    ziDoubleType* Double;  
    ziIntegerType* Integer;  
    TreeChange* Tree;  
    ByteArrayData* ByteArray;  
    ScopeWave* Wave;  
    uint64_t alignment;  
} ziEvent::Val;
```

Data Fields

- void* Void
- DemodSample* SampleDemod
- AuxInSample* SampleAuxIn
- DIOSample* SampleDIO
- ziDoubleType* Double
- ziIntegerType* Integer
- TreeChange* Tree
- ByteArrayData* ByteArray
- ScopeWave* Wave
- uint64_t alignment

union ziEvent::Val

```
typedef union ziEvent::Val {  
    void* Void;  
    DemodSample* SampleDemod;  
    AuxInSample* SampleAuxIn;  
    DIOSample* SampleDIO;  
    ziDoubleType* Double;  
    ziIntegerType* Integer;  
    TreeChange* Tree;  
    ByteArrayData* ByteArray;  
    ScopeWave* Wave;  
    uint64_t alignment;  
} ziEvent::Val;
```

Data Fields

- void* Void
- DemodSample* SampleDemod
- AuxInSample* SampleAuxIn
- DIOSample* SampleDIO
- ziDoubleType* Double
- ziIntegerType* Integer
- TreeChange* Tree
- ByteArrayData* ByteArray
- ScopeWave* Wave
- uint64_t alignment

Enumeration Type Documentation

Defines return value for all ziAPI functions. Divided into 3 regions: info, warning and error.

Enumerator:

- ZI_INFO_BASE
- ZI_INFO_SUCCESS
Success (no error)
- ZI_INFO_MAX
- ZI_WARNING_BASE
- ZI_WARNING_GENERAL
Warning (general);.
- ZI_WARNING_UNDERRUN
FIFO Underrun.
- ZI_WARNING_OVERFLOW
FIFO Overflow.
- ZI_WARNING_NOTFOUND
Value or Node not found.
- ZI_WARNING_NO_ASYNC
Async command executed in sync mode (will be no async reply)
- ZI_WARNING_MAX
- ZI_ERROR_BASE
- ZI_ERROR_GENERAL
Error (general)
- ZI_ERROR_USB
USB Communication failed.
- ZI_ERROR_MALLOC
Memory allocation failed.
- ZI_ERROR_MUTEX_INIT
Unable to initialize mutex.
- ZI_ERROR_MUTEX_DESTROY
Unable to destroy mutex.
- ZI_ERROR_MUTEX_LOCK
Unable to lock mutex.
- ZI_ERROR_MUTEX_UNLOCK
Unable to unlock mutex.
- ZI_ERROR_THREAD_START

- Unable to start thread.
- ZI_ERROR_THREAD_JOIN
Unable to join thread.
- ZI_ERROR_SOCKET_INIT
Can't initialize socket.
- ZI_ERROR_SOCKET_CONNECT
Unable to connect socket.
- ZI_ERROR_HOSTNAME
Hostname not found.
- ZI_ERROR_CONNECTION
Connection invalid.
- ZI_ERROR_TIMEOUT
Command timed out.
- ZI_ERROR_COMMAND
Command internally failed.
- ZI_ERROR_SERVER_INTERNAL
Command failed in server.
- ZI_ERROR_LENGTH
Provided Buffer length is too small.
- ZI_ERROR_FILE
Can't open file or read from it.
- ZI_ERROR_DUPLICATE
There is already a similar entry.
- ZI_ERROR_READONLY
Attempt to set a read-only node.
- ZI_ERROR_DEVICE_NOT_VISIBLE
Device is not visible to the server.
- ZI_ERROR_DEVICE_IN_USE
Device is already connected by a different server.
- ZI_ERROR_DEVICE_INTERFACE
Device does currently not support the specified interface.
- ZI_ERROR_DEVICE_CONNECTION_TIMEOUT
Device connection timeout.
- ZI_ERROR_DEVICE_DIFFERENT_INTERFACE
Device already connected over a different Interface.
- ZI_ERROR_DEVICE_NEEDS_FW_UPGRADE
Device needs FW upgrade.

- `ZI_ERROR_ZIEVENT_DATATYPE_MISMATCH`
Trying to get data from a poll event with wrong target data type.
- `ZI_ERROR_DEVICE_NOT_FOUND`
Device not found.
- `ZI_ERROR_NOT_SUPPORTED`
Provided arguments are not supported for the command.
- `ZI_ERROR_TOO_MANY_CONNECTIONS`
Connection invalid.
- `ZI_ERROR_NOT_ON_HF2`
Command not supported on HF2.
- `ZI_ERROR_MAX`

Enumerates all types that data in a [ZIEvent](#) may have.

Enumerator:

- `ZI_VALUE_TYPE_NONE`
No data type, event is invalid.
- `ZI_VALUE_TYPE_DOUBLE_DATA`
`ZIDoubleData` type. Use the `ZIEvent.value.doubleData` pointer to read the data of the event.
- `ZI_VALUE_TYPE_INTEGER_DATA`
`ZIIntegerData` type. Use the `ZIEvent.value.integerData` pointer to read the data of the event.
- `ZI_VALUE_TYPE_DEMOD_SAMPLE`
[ZIDemodSample](#) type. Use the `ZIEvent.value.demodSample` pointer to read the data of the event.
- `ZI_VALUE_TYPE_SCOPE_WAVE_OLD`
[ScopeWave](#) type, used in v1 compatibility mode. use the `ZIEvent.value.scopeWaveOld` pointer to read the data of the event.
- `ZI_VALUE_TYPE_AUXIN_SAMPLE`
[ZIAuxInSample](#) type. Use the `ZIEvent.value.auxInSample` pointer to read the data of the event.
- `ZI_VALUE_TYPE_DIO_SAMPLE`
[ZIDIOSample](#) type. Use the `ZIEvent.value.dioSample` pointer to read the data of the event.
- `ZI_VALUE_TYPE_BYTE_ARRAY`
[ZIByteArray](#) type. Use the `ZIEvent.value.byteArray` pointer to read the data of the event.
- `ZI_VALUE_TYPE_PWA_WAVE`
[ZIPWAWave](#) type. Use the `ZIEvent.value.pwaWave` pointer to read the data of the event.
- `ZI_VALUE_TYPE_TREE_CHANGE_DATA_OLD`
[TreeChange](#) type - a list of added or removed nodes, used in v1 compatibility mode. Use the `ZIEvent.value.treeChangeDataOld` pointer to read the data of the event.
- `ZI_VALUE_TYPE_DOUBLE_DATA_TS`
[ZIDoubleDataTS](#) type. Use the `ZIEvent.value.doubleDataTS` pointer to read the data of the event.
- `ZI_VALUE_TYPE_INTEGER_DATA_TS`
[ZIIntegerDataTS](#) type. Use the `ZIEvent.value.integerDataTS` pointer to read the data of the event.
- `ZI_VALUE_TYPE_COMPLEX_DATA`

- [ZIComplexData](#) type. Use the `ZIEvent.value.complexData` pointer to read the data of the event.
- `ZI_VALUE_TYPE_SCOPE_WAVE`

[ZIScopeWave](#) type. Use the `ZIEvent.value.scopeWave` pointer to read the data of the event.
- `ZI_VALUE_TYPE_SCOPE_WAVE_EX`

[ZIScopeWaveEx](#) type. Use the `ZIEvent.value.scopeWaveEx` pointer to read the data of the event.
- `ZI_VALUE_TYPE_BYTE_ARRAY_TS`

[ZIByteArrayTS](#) type. Use the `ZIEvent.value.byteArrayTS` pointer to read the data of the event.
- `ZI_VALUE_TYPE_CNT_SAMPLE`

[ZICntSample](#) type. Use the `ZIEvent.value.cntSample` pointer to read the data of the event.
- `ZI_VALUE_TYPE_TRIG_SAMPLE`

[ZITrigSample](#) type. Use the `ZIEvent.value.trigSample` pointer to read the data of the event.
- `ZI_VALUE_TYPE_TREE_CHANGE_DATA`

[ZITreeChangeData](#) type - a list of added or removed nodes. Use the `ZIEvent.value.treeChangeData` pointer to read the data of the event.
- `ZI_VALUE_TYPE_ASYNC_REPLY`

`ZIAsyncReply` type. Use the `ZIEvent.value.asyncReply` pointer to read the data of the event.
- `ZI_VALUE_TYPE_SWEEPER_WAVE`

`ZISweeperWave` type. Use the `ZIEvent.value.sweeperWave` pointer to read the data of the event.
- `ZI_VALUE_TYPE_SPECTRUM_WAVE`

`ZISpectrumWave` type. Use the `ZIEvent.value.spectrumWave` pointer to read the data of the event.
- `ZI_VALUE_TYPE_ADVISOR_WAVE`

`ZIAdvisorWave` type. Use the `ZIEvent.value.advisorWave` pointer to read the data of the event.
- `ZI_VALUE_TYPE_VECTOR_DATA`

[ZIVectorData](#) type. Use the `ZIEvent.value.vectorData` pointer to access the data of the event.
- `ZI_VALUE_TYPE_IMPEDANCE_SAMPLE`

[ZIImpedanceSample](#) type. Use the `ZIEvent.value.impedanceSample` pointer to access the data of the event.

Defines the actions that are performed on a tree, as returned in the [ZITreeChangeData::action](#) or [ZITreeChangeDataOld::action](#).

Enumerator:

- `ZI_TREE_ACTION_REMOVE`
A node has been removed.
- `ZI_TREE_ACTION_ADD`
A node has been added.
- `ZI_TREE_ACTION_CHANGE`
A node has been changed.

Enumerates the bits set in an [ZIImpedanceSample](#)'s flags.

Enumerator:

- ZI_IMP_FLAGS_NONE
- ZI_IMP_FLAGS_VALID_INTERNAL
Internal calibration is applied.
- ZI_IMP_FLAGS_VALID_USER
User compensation is applied.
- ZI_IMP_FLAGS_AUTORANGE_GATING
Reserved for future use.
- ZI_IMP_FLAGS_OVERFLOW_VOLTAGE
Overflow on voltage input.
- ZI_IMP_FLAGS_OVERFLOW_CURRENT
Overflow on current input.
- ZI_IMP_FLAGS_UNDERFLOW_VOLTAGE
Underflow on voltage input.
- ZI_IMP_FLAGS_UNDERFLOW_CURRENT
Underflow on current input.
- ZI_IMP_FLAGS_FREQ_EXACT
Reserved for future use.
- ZI_IMP_FLAGS_FREQ_INTERPOLATION
Reserved for future use.
- ZI_IMP_FLAGS_FREQ_EXTRAPOLATION
Reserved for future use.
- ZI_IMP_FLAGS_LOWDUT2T
LowDUT impedance detected.
- ZI_IMP_FLAGS_SUPPRESSION_PARAM0
Suppression of first parameter PARAM0.
- ZI_IMP_FLAGS_SUPPRESSION_PARAM1
Suppression of second parameter PARAM1.
- ZI_IMP_FLAGS_FREQLIMIT_RANGE_VOLTAGE
Reserved for future use.
- ZI_IMP_FLAGS_FREQLIMIT_RANGE_CURRENT
Frequency bigger than the frequency limit of active current input range.
- ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM0
Strong compensation detected on PARAM0.
- ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM1

Strong compensation detected on PARAM1.

- ZI_IMP_FLAGS_NEGATIVE_QFACTOR
Non-reasonable values for Q/D measurement.
- ZI_IMP_FLAGS_BWC_BIT0
Reserved for future use.
- ZI_IMP_FLAGS_BWC_BIT1
Reserved for future use.
- ZI_IMP_FLAGS_BWC_BIT2
Reserved for future use.
- ZI_IMP_FLAGS_BWC_BIT3
Reserved for future use.
- ZI_IMP_FLAGS_BWC_MASK
Reserved for future use.
- ZI_IMP_FLAGS_OPEN_DETECTION
Open detected on 4T measurement.
- ZI_IMP_FLAGS_OVERFLOW_SIGIN0
Overflow on sign0.
- ZI_IMP_FLAGS_OVERFLOW_SIGIN1
Overflow on sign1.
- ZI_IMP_FLAGS_MODEL_MASK
Model selected for the measurement.

Enumerates all the types that a `ZIVectorData::elementType` may have.

Enumerator:

- `ZI_VECTOR_ELEMENT_TYPE_UINT8`
- `ZI_VECTOR_ELEMENT_TYPE_UINT16`
- `ZI_VECTOR_ELEMENT_TYPE_UINT32`
- `ZI_VECTOR_ELEMENT_TYPE_UINT64`
- `ZI_VECTOR_ELEMENT_TYPE_FLOAT`
- `ZI_VECTOR_ELEMENT_TYPE_DOUBLE`
- `ZI_VECTOR_ELEMENT_TYPE_ASCII`
NULL-terminated string.

Enumerator:

- ZI_API_VERSION_0
- ZI_API_VERSION_1
- ZI_API_VERSION_4
- ZI_API_VERSION_5
- ZI_API_VERSION_6
- ZI_API_VERSION_MAX

Defines the values of the flags used in [ziAPIListNodes](#).

Enumerator:

- **ZI_LIST_NODES_NONE**
Default, return a simple listing of the given node immediate descendants.
- **ZI_LIST_NODES_RECURSIVE**
List the nodes recursively.
- **ZI_LIST_NODES_ABSOLUTE**
Return absolute paths.
- **ZI_LIST_NODES_LEAFONLY**
Return only leaf nodes, which means the nodes at the outermost level of the tree.
- **ZI_LIST_NODES_SETTINGSONLY**
Return only nodes which are marked as setting.
- **ZI_LIST_NODES_STREAMINGONLY**
Return only streaming nodes (nodes that can be pushed from the device at a high data rate)
- **ZI_LIST_NODES_SUBSCRIBEDONLY**
Return only nodes that are subscribed to in the API session.
- **ZI_LIST_NODES_BASECHANNEL**
- **ZI_LIST_NODES_GETONLY**

Defines the flags returned in the chunk header for all modules.

Enumerator:

- `ZI_CHUNK_HEADER_FLAG_FINISHED`
Indicates that the chunk data is complete. This flag will be set if data is read out from the module before the measurement (e.g. sweep) has finished.
- `ZI_CHUNK_HEADER_FLAG_ROLLMODE`
Unused.
- `ZI_CHUNK_HEADER_FLAG_DATALOSS`
Indicates that dataloss has occurred.
- `ZI_CHUNK_HEADER_FLAG_VALID`
Indicates that the data is valid.
- `ZI_CHUNK_HEADER_FLAG_DATA`
Indicates whether the chunk contains data (opposite to setting).
- `ZI_CHUNK_HEADER_FLAG_DISPLAY`
Internal use only.
- `ZI_CHUNK_HEADER_FLAG_FREQDOMAIN`
chunk contains frequency domain data, opposite to time domain.
- `ZI_CHUNK_HEADER_FLAG_SPECTRUM`
chunk recorded in spectrum mode.
- `ZI_CHUNK_HEADER_FLAG_OVERLAPPED`
chunk results overlap with neighboring chunks, see spectrum.
- `ZI_CHUNK_HEADER_FLAG_ROWFINISHED`
indicates that the current row is finished - useful for row first averaging.
- `ZI_CHUNK_HEADER_FLAG_ONGRIDSAMPLING`
exact sampling was used.
- `ZI_CHUNK_HEADER_FLAG_ROWREPETITION`
row first averaging is enabled.
- `ZI_CHUNK_HEADER_FLAG_PREVIEW`
chunk contains preview (fft with less points).

Defines flags returned in the chunk header that only apply for certain modules.

Enumerator:

- ZI_CHUNK_HEADER_MODULE_FLAGS_WINDOW
FFT Window used in the Data Acquisition Module: 0 - Rectangular, 1 - Hann, 2 - Hamming, 3 - Blackmanharris, 4 - Exponential, 5 - Cosine, 6 - CosineSquare.

Enumerator:

- ZI_VECTOR_WRITE_STATUS_IDLE
- ZI_VECTOR_WRITE_STATUS_PENDING

TREE_ACTION defines the values for the [TreeChange::Action](#) Variable.

Enumerator:

- TREE_ACTION_REMOVE
a tree has been removed
- TREE_ACTION_ADD
a tree has been added
- TREE_ACTION_CHANGE
a tree has changed

Function Documentation

ziAPIInit

ZIResult_enum ziAPIInit (**ZIConnection*** conn)

Initializes a **ZIConnection** structure.

This function initializes the structure so that it is ready to connect to Data Server. It allocates memory and sets up the infrastructure needed.

Parameters:

[out] conn
Pointer to **ZIConnection** that is to be initialized

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_MALLOC on memory allocation failure
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDestroy](#), [ziAPIConnect](#), [ziAPIDisconnect](#)

See [Connection](#) for an example

ziAPIDestroy

ZIResult_enum ziAPIDestroy (ZIConnection conn)

Destroys a [ZIConnection](#) structure.

This function frees all memory that has been allocated by [ziAPIInit](#). If it is called with an uninitialized [ZIConnection](#) struct it may result in segmentation faults as well when it is called with a struct for which [ZIAPIDestroy](#) already has been called.

Parameters:

[in] conn

Pointer to [ZIConnection](#) struct that has to be destroyed

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIInit](#), [ziAPIConnect](#), [ziAPIDisconnect](#)

See [Connection](#) for an example

ziAPIConnect

ZIResult_enum ziAPIConnect ([ZIConnection](#) conn, const char* hostname, uint16_t port)

Connects the [ZIConnection](#) to Data Server.

Connects to Data Server using a [ZIConnection](#) and prepares for data exchange. For most cases it is enough to just give a reference to the connection and give NULL for hostname and 0 for the port, so it connects to localhost on the default port.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the connection should be established

[in] hostname

Name of the Host to which it should be connected, if NULL "localhost" will be used as default

[in] port

The Number of the port to connect to. If 0, default port of the local Data Server will be used (8005)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_HOSTNAME if the given host name could not be found
- ZI_ERROR_SOCKET_CONNECT if no connection could be established
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_SOCKET_INIT if initialization of the socket failed
- ZI_ERROR_CONNECTION when the Data Server didn't return the correct answer
- ZI_ERROR_TIMEOUT when initial communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDisconnect](#), [ziAPIInit](#), [ziAPIDestroy](#)

See [Connection](#) for an example

ziAPIDisconnect

ZIResult_enum `ziAPIDisconnect ( ZIConnection conn )`

Disconnects an established connection.

Disconnects from Data Server. If the connection has not been established and the function is called it returns without doing anything.

Parameters:

[in] conn
Pointer to ZIConnection to be disconnected

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnect](#), [ziAPIInit](#), [ziAPIDestroy](#)

See [Connection](#) for an example

ziAPIListImplementations

ZIResult_enum ziAPIListImplementations (char* implementations, uint32_t bufferSize)

Returns the list of supported implementations.

Returned names are defined by implementations in the linked library and may change depending on software version.

Parameters:

[out] implementations

Pointer to a buffer receiving a newline-delimited list of the names of all the supported ziAPI implementations. The string is zero-terminated.

[in] bufferSize

The size of the buffer assigned to the implementations parameter

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_LENGTH if the length of the char-buffer given by MaxLen is too small for all elements
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#)

ziAPIConnectEx

ZIResult_enum ziAPIConnectEx (**ZIConnection** conn, const char* hostname, uint16_t port, ZIAPIVersion_enum apiLevel, const char* implementation)

Connects to Data Server and enables extended ziAPI.

With apiLevel=ZI_API_VERSION_1 and implementation=NULL, this call is equivalent to plain [ziAPIConnect](#). With other version and implementation values enables corresponding ziAPI extension and connection using different implementation.

Parameters:

[in] conn

Pointer to the ZIConnection with which the connection should be established

[in] hostname

Name of the host to which it should be connected, if NULL "localhost" will be used as default

[in] port

The number of the port to connect to. If 0 the port of the local Data Server will be used

[in] apiLevel

Specifies the ziAPI compatibility level to use for this connection (1 or 4).

[in] implementation

Specifies implementation to use for a connection, must be one of the returned by ziAPIListImplementations or NULL to select default implementation

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_HOSTNAME if the given host name could not be found
- ZI_ERROR_SOCKET_CONNECT if no connection could be established
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_SOCKET_INIT if initialization of the socket failed
- ZI_ERROR_CONNECTION when the Data Server didn't return the correct answer or requested implementation is not found or doesn't support requested ziAPI level
- ZI_ERROR_TIMEOUT when initial communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIListImplementations](#), [ziAPIConnect](#), [ziAPIDisconnect](#), [ziAPIInit](#), [ziAPIDestroy](#), [ziAPIGetConnectionVersion](#)

See [Connection](#) for an example

ziAPIGetConnectionAPILevel

ZIResult_enum ziAPIGetConnectionAPILevel (**ZIConnection** conn, ZIAPIVersion_enum* apiLevel)

Returns ziAPI level used for the connection conn.

Parameters:

[in] conn

Pointer to ZIConnection

[out] apiLevel

Pointer to preallocated ZIAPIVersion_enum, receiving the ziAPI level

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION if level can not be determined due to conn is not connected

See Also:

[ziAPIConnectEx](#), [ziAPIGetVersion](#), [ziAPIGetRevision](#)

ziAPIGetVersion

ZIResult_enum ziAPIGetVersion (const char** version)

Retrieves the release version of ziAPI.

Sets the passed pointer to point to the null-terminated release version string of ziAPI.

Parameters:

[in] version

Pointer to const char pointer.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#), [ziAPIGetRevision](#), [ziAPIGetConnectionAPILevel](#)

ziAPIGetRevision

ZIResult_enum ziAPIGetRevision (unsigned int* revision)

Retrieves the revision of ziAPI.

Sets an unsigned int with the revision (build number) of the ziAPI you are using.

Parameters:

[in] revision

Pointer to an unsigned int to fill up with the revision.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectEx](#), [ziAPIGetVersion](#), [ziAPIGetConnectionAPILevel](#)

ziAPIListNodes

ZIResult_enum ziAPIListNodes (**ZIConnection** conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child nodes found at the specified path.

This function returns a list of node names found at the specified path. The path may contain wildcards so that the returned nodes do not necessarily have to have the same parents. The list is returned in a null-terminated char-buffer, each element delimited by a newline. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be ZIResult_enum::ZI_LENGTH.

Parameters:

[in] conn

Pointer to the ZIConnection for which the node names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with newline-delimited list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer used for the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in [ZIListNodes_enum](#).

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds [MAX_PATH_LEN](#) or the length of the char-buffer for the nodes given by bufferSize is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Tree Listing](#) for an example

See Also:

[ziAPIUpdate](#)

ziAPIListNodesJSON

ZIResult_enum ziAPIListNodesJSON (**ZIConnection** conn, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child nodes found at the specified path.

This function returns a list of node names found at the specified path, formatted as JSON. The path may contain wildcards so that the returned nodes do not necessarily have to have the same parents. The list is returned in a null-terminated char-buffer. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be ZIResult_enum::ZI_LENGTH.

Parameters:

[in] conn

Pointer to the ZIConnection for which the node names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with JSON-formatted list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer used for the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in [ZIListNodes_enum](#).

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds [MAX_PATH_LEN](#) or the length of the char-buffer for the nodes given by bufferSize is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Tree Listing](#) for an example

See Also:

[ziAPIUpdate](#)

ziAPIUpdateDevices

[ZIResult_enum](#) ziAPIUpdateDevices ([ZIConnection](#) conn)

Search for the newly connected devices and update the tree.

This function forces the Data Server to search for newly connected devices and to connect to run them

Parameters:

[in] conn
Pointer to ZIConnection

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIListNodes](#)

ziAPIConnectDevice

ZIResult_enum ziAPIConnectDevice (**ZIConnection** conn, const char* deviceSerial, const char* deviceInterface, const char* interfaceParams)

Connect a device to the server.

This function connects a device with deviceSerial via the specified deviceInterface for use with the server.

Parameters:

[in] conn

Pointer to the ZIConnection with which the connection should be established

[in] deviceSerial

The serial of the device to connect to, e.g., dev2100

[in] deviceInterface

The interface to use for the connection, e.g., USB|1GbE

[in] interfaceParams

Parameters for interface configuration (currently reserved for future use, NULL may be specified).

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDisconnectDevice](#), [ziAPIConnect](#), [ziAPIDisconnect](#), [ziAPIInit](#)

ziAPIDisconnectDevice

ZIResult_enum `ziAPIDisconnectDevice ( ZIConnection conn, const char* deviceSerial )`

Disconnect a device from the server.

This function disconnects a device specified by deviceSerial from the server.

Parameters:

[in] conn

Pointer to the [ZIConnection](#) with which the connection should be established

[in] deviceSerial

The serial of the device to connect to, e.g., dev2100

Returns:

- [ZI_INFO_SUCCESS](#) on success
- [ZI_ERROR_TIMEOUT](#) when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIConnectDevice](#), [ziAPIConnect](#), [ziAPIDisconnect](#), [ziAPIInit](#)

ziAPIGetValueD

ZIResult_enum ziAPIGetValueD (**ZIConnection** conn, const char* path, ZIDoubleData* value)

gets the double-type value of the specified node

This function retrieves the numerical value of the specified node as an double-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a double in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2018] Zurich Instruments AG
```

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
```

```
#include "ziAPI.h"
#include "ziUtils.h"
```

```
void setDemodRate(ZIConnection conn, const char** deviceId, uint32_t index,
ZIDoubleData rate) {
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPISetValueD(conn, nodePath, rate));
}
```

```
ZIDoubleData getDemodRate(ZIConnection conn, const char** deviceId, uint32_t index) {
    ZIDoubleData rate;
```

```
char nodePath[1024];
snprintf(nodePath, sizeof(nodePath), "%s/demods/%d/rate", *deviceId, index);
checkError(ziAPIGetValueD(conn, nodePath, &rate));
return rate;
}

int main() {
    // The device address of the device to run the example on.
    char deviceAddress[] = "dev2006";
    // The maximum API Level supported by this example.
    // Please use ZI_API_VERSION_1 if using an HF2 Instrument.
    ZIAPIVersion_enum apiLevel = ZI_API_VERSION_6;

    // The ZIConnection is actually a pointer.
    ZIConnection conn;
    if (isError(ziAPIInit(&conn))) {
        return 1;
    }

    ziAPISetDebugLevel(0);
    ziAPIWriteDebugLog(0, "Logging enabled.");

    const char *deviceId;
    if (!ziCreateAPISession(conn, deviceAddress, apiLevel, &deviceId)) {
        try {
            ziApiServerVersionCheck(conn);

            // Set a device configuration.
            uint32_t index = 0;
            ZIDoubleData rate = 10e3;
            setDemodRate(conn, &deviceId, index, rate);

            // Read it back.
            rate = getDemodRate(conn, &deviceId, index);
            std::cout << "[INFO] " << "Device " << deviceId << " demod " << index << " has
rate " << rate << ".\n";

        } catch (std::runtime_error& e) {
            char extErrorMessage[1024] = "";
            ziAPIGetLastError(conn, extErrorMessage, 1024);
            fprintf(stderr, "[ERROR] %s\ndetails: `%s`\n", e.what(), extErrorMessage);
        } catch (...) {
            // Ensure all exceptions are caught.
            fprintf(stderr, "[ERROR] Unexpected error\n.");
        }
        ziAPIDisconnect(conn);
    } else {
        char extErrorMessage[1024] = "";
        ziAPIGetLastError(conn, extErrorMessage, 1024);
        fprintf(stderr, "[ERROR] Details: `%s`\n", extErrorMessage);
    }
    ziAPIDestroy(conn);

    return 0;
}
```

See Also:

[ziAPISetValueD](#), [ziAPIGetValueAsPollData](#)

ziAPIGetComplexData

ZIResult_enum ziAPIGetComplexData (**ZIConnection** conn, const char* path, **ZIDoubleData*** real, **ZIDoubleData*** imag)

gets the complex double-type value of the specified node

This function retrieves the numerical value of the specified node as an complex double-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] real

Pointer to a double in which the real value should be written

[out] imag

Pointer to a double in which the imaginary value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2018] Zurich Instruments AG
```

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
```

```
#include "ziAPI.h"
#include "ziUtils.h"
```

```
void setDemodRate(ZIConnection conn, const char** deviceId, uint32_t index,
ZIDoubleData rate) {
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "/%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPISetValueD(conn, nodePath, rate));
}
```

```
}

ZIDoubleData getDemodRate(ZIConnection conn, const char** deviceId, uint32_t index) {
    ZIDoubleData rate;
    char nodePath[1024];
    snprintf(nodePath, sizeof(nodePath), "%s/demods/%d/rate", *deviceId, index);
    checkError(ziAPIGetValueD(conn, nodePath, &rate));
    return rate;
}

int main() {
    // The device address of the device to run the example on.
    char deviceAddress[] = "dev2006";
    // The maximum API Level supported by this example.
    // Please use ZI_API_VERSION_1 if using an HF2 Instrument.
    ZIAPIVersion_enum apiLevel = ZI_API_VERSION_6;

    // The ZIConnection is actually a pointer.
    ZIConnection conn;
    if (isError(ziAPIInit(&conn))) {
        return 1;
    }

    ziAPISetDebugLevel(0);
    ziAPIWriteDebugLog(0, "Logging enabled.");

    const char *deviceId;
    if (!ziCreateAPISession(conn, deviceAddress, apiLevel, &deviceId)) {
        try {
            ziApiServerVersionCheck(conn);

            // Set a device configuration.
            uint32_t index = 0;
            ZIDoubleData rate = 10e3;
            setDemodRate(conn, &deviceId, index, rate);

            // Read it back.
            rate = getDemodRate(conn, &deviceId, index);
            std::cout << "[INFO] " << "Device " << deviceId << " demod " << index << " has
rate " << rate << ".\n";

        } catch (std::runtime_error& e) {
            char extErrorMessage[1024] = "";
            ziAPIGetLastError(conn, extErrorMessage, 1024);
            fprintf(stderr, "[ERROR] %s\ndetails: `%s`\n.", e.what(), extErrorMessage);
        } catch (...) {
            // Ensure all exceptions are caught.
            fprintf(stderr, "[ERROR] Unexpected error\n.");
        }
        ziAPIDisconnect(conn);
    } else {
        char extErrorMessage[1024] = "";
        ziAPIGetLastError(conn, extErrorMessage, 1024);
        fprintf(stderr, "[ERROR] Details: `%s`\n", extErrorMessage);
    }
    ziAPIDestroy(conn);

    return 0;
}
```

See Also:

[ziAPISetComplexData](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueI

ZIResult_enum ziAPIGetValueI (**ZIConnection** conn, const char* path, ZIIntegerData* value)

gets the integer-type value of the specified node

This function retrieves the numerical value of the specified node as an integer-type value. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to an 64bit integer in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueI](#), [ziAPIGetValueAsPollData](#)

ziAPIGetDemodSample

ZIResult_enum ziAPIGetDemodSample (**ZIConnection** conn, const char* path, **ZIDemodSample*** value)

Gets the demodulator sample value of the specified node.

This function retrieves the value of the specified node as an **DemodSample** struct. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to paths matching DEMODS/[0-9]+/SAMPLE.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a **ZIDemodSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetSampleS(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIDemodSample DemodSample;

    if ((RetVal = ziAPIGetDemodSample(Conn,
                                     "/dev1046/demods/0/sample",
                                     &DemodSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
```



```
        printf("TS = %f, X=%f, Y=%f\n",
               (float)DemodSample.timeStamp,
               DemodSample.x,
               DemodSample.y);
    }
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetDIOSample

ZIResult_enum ziAPIGetDIOSample (**ZIConnection** conn, const char* path, **ZIDIOSample*** value)

Gets the Digital I/O sample of the specified node.

This function retrieves the newest available DIO sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to nodes ending in "/DIOS/[0-9]+/INPUT".

Parameters:

[in] conn

Pointer to the ZIConnection with which the value should be retrieved

[in] path

Path to the node holding the value

[out] value

Pointer to a **ZIDIOSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetDIOSample(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIDIOSample DIOSample;

    if ((RetVal = ziAPIGetDIOSample(Conn,
                                    "/dev1046/dios/0/output",
                                    &DIOSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
```

```
        printf("TS = %f, bits=%08x\n",  
              (float)DIOSample.timeStamp,  
              DIOSample.bits);  
    }  
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetAuxInSample

ZIResult_enum ziAPIGetAuxInSample (**ZIConnection** conn, const char* path, **ZIAuxInSample*** value)

gets the AuxIn sample of the specified node

This function retrieves the newest available AuxIn sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path). This function is only applicable to nodes ending in "/AUXINS/[0-9]+/SAMPLE".

Parameters:

[in] conn

Pointer to the ziConnection with which the Value should be retrieved

[in] path

Path to the Node holding the value

[out] value

Pointer to an **ZIAuxInSample** struct in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void GetSampleAuxIn(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    ZIAuxInSample AuxInSample;

    if ((RetVal = ziAPIGetAuxInSample(Conn,
                                     "/dev1046/auxins/0/sample",
                                     &AuxInSample)) != ZI_INFO_SUCCESS) {
        ziAPIGetError(RetVal, &ErrBuffer, NULL);
        fprintf(stderr, "Error, can't get Parameter: %s\n", ErrBuffer);
    }
}
```

```
    } else {  
        printf("TS = %f, ch0=%f, ch1=%f\n",  
              (float)AuxInSample.timeStamp,  
              AuxInSample.ch0,  
              AuxInSample.ch1);  
    }  
}
```

See Also:

[ziAPIGetValueAsPollData](#)

ziAPIGetValueB

ZIResult_enum ziAPIGetValueB (**ZIConnection** conn, const char* path, unsigned char* buffer, unsigned int* length, unsigned int bufferSize)

gets the Bytearray value of the specified node

This function retrieves the newest available DIO sample from the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved data in

[out] length

Pointer to an unsigned int to store the length of data in. if an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void PrintVersion(ZIConnection Conn) {
    ZIResult_enum RetVal;
    char* ErrBuffer;
```

```
const char* Path = "ZI/ABOUT/VERSION";
unsigned char Buffer[0xff];
unsigned int Length;

if ((RetVal = ziAPIGetValueB(Conn,
                             Path,
                             Buffer,
                             &Length,
                             sizeof(Buffer) - 1)) != ZI\_INFO\_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't get value: %s.\n", ErrBuffer);
} else {
    Buffer[Length] = 0;
    printf("%s=\"%s\"\n", Path, Buffer);
}
}
```

See Also:

[ziAPISetValueB](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueString

ZIResult_enum ziAPIGetValueString (**ZIConnection** conn, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize)

gets a null-terminated string value of the specified node

This function retrieves the newest string value for the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueString](#), [ziAPIGetValueAsPollData](#)

ziAPIGetValueStringUnicode

ZIResult_enum ziAPIGetValueStringUnicode (**ZIConnection** conn, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize)

gets a null-terminated string value of the specified node

This function retrieves the newest unicode string value for the specified node. The value first found is returned if more than one value is available (a wildcard is used in the path).

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] path

Path to the Node holding the value

[out] wbuffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPISetValueStringUnicode](#), [ziAPIGetValueAsPollData](#)

ziAPISetValueD

ZIResult_enum ziAPISetValueD (**ZIConnection** conn, const char* path, ZIDoubleData value)

asynchronously sets a double-type value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set.

[in] path

Path to the Node(s) for which the value(s) will be set to Value.

[in] value

The double-type value that will be written to the node(s).

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueD](#). [ziAPISyncSetValueD](#)

ziAPISetComplexData

ZIResult_enum ziAPISetComplexData (**ZIConnection** conn, const char* path, ZIDoubleData real, ZIDoubleData imag)

asynchronously sets a double-type complex value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to the complex value (real, imag). More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set. If the node does not support complex values only the real value will be updated.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set.

[in] path

Path to the Node(s) for which the value(s) will be set to Value.

[in] real

The real value that will be written to the node(s).

[in] imag

The imag value that will be written to the node(s).

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetComplexData](#). [ziAPISyncSetComplexData](#)

ziAPISetValue1

ZIResult_enum ziAPISetValue1 (**ZIConnection** conn, const char* path, ZIntegerData value)

asynchronously sets an integer-type value to one or more nodes specified in a path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] value

The int-type value that will be written to the node(s)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValue1](#). [ziAPISyncSetValue1](#)

ziAPISetValueB

ZIResult_enum ziAPISetValueB (**ZIConnection** conn, const char* path, unsigned char* buffer, unsigned int length)

asynchronously sets the binary-type value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] buffer

Pointer to the byte array with the data

[in] length

Length of the data in the buffer

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void ProgramCPU(ZIConnection Conn,
                unsigned char* Buffer,
                int Len) {
    ZIResult_enum RetVal;
    char* ErrBuffer;

    if ((RetVal = ziAPISetValueB(Conn,
```

```
        "/dev1046/cpus/0/program",
        Buffer,
        Len)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
}
}
```

See Also:

[ziAPIGetValueB](#). [ziAPISyncSetValueB](#)

ziAPISetValueString

ZIResult_enum ziAPISetValueString (**ZIConnection** conn, const char* path, const char* str)

asynchronously sets a string value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] str

Pointer to a null terminated string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueString](#). [ziAPISyncSetValueString](#)

ziAPISetValueStringUnicode

ZIResult_enum ziAPISetValueStringUnicode (**ZIConnection** conn, const char* path, const wchar_t* wstr)

asynchronously sets a unicode encoded string value of one or more nodes specified in the path

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. The function sets the value asynchronously which means that after the function returns you have no security to which value it is finally set nor at what point in time it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] wstr

Pointer to a null terminated unicode string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred.
- ZI_ERROR_READONLY on attempt to set a read-only node.
- ZI_ERROR_COMMAND on an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT when communication timed out.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueStringUnicode](#). [ziAPISyncSetValueStringUnicode](#)

ziAPISyncSetValueD

ZIResult_enum ziAPISyncSetValueD (**ZIConnection** conn, const char* path, ZIDoubleData* value)

synchronously sets a double-type value to one or more nodes specified in the path

This function sets the values of the nodes specified in path to Value. More than one value can be set if a wildcard is used. The function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set to value

[in] value

Pointer to a double-type containing the value to be written. When the function returns value holds the effectively written value.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueD](#), [ziAPISetValueD](#)

ziAPISyncSetValueI

ZIResult_enum ziAPISyncSetValueI (**ZIConnection** conn, const char* path, **ZIIntegerData*** value)

synchronously sets an integer-type value to one or more nodes specified in a path

This function sets the values of the nodes specified in path to value. More than one value can be set if a wildcard is used. The function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the node(s) for which the value(s) will be set

[in] value

Pointer to a int-type containing then value to be written. when the function returns value holds the effectively written value.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueI](#), [ziAPISetValueI](#)

ziAPISyncSetValueB

ZIResult_enum ziAPISyncSetValueB ([ZIConnection](#) conn, const char* path, uint8_t* buffer, uint32_t* length, uint32_t bufferSize)

Synchronously sets the binary-type value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set and to which value it is set.

Parameters:

[in] conn

Pointer to the [ziConnection](#) for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in] buffer

Pointer to the byte array with the data

[in] length

Length of the data in the buffer

[in] bufferSize

Length of the data in the buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueB](#), [ziAPISetValueB](#)

ziAPISyncSetValueString

ZIResult_enum ziAPISyncSetValueString (**ZIConnection** conn, const char* path, const char* str)

Synchronously sets a string value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in/ out] str

Pointer to a null terminated string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueString](#), [ziAPISetValueString](#)

ziAPISyncSetValueStringUnicode

ZIResult_enum ziAPISyncSetValueStringUnicode (**ZIConnection** conn, const char* path, const wchar_t* wstr)

Synchronously sets a unicode string value of one or more nodes specified in the path.

This function sets the values at the nodes specified in a path. More than one value can be set if a wildcard is used. This function sets the value synchronously. After returning you know that it is set.

Parameters:

[in] conn

Pointer to the ziConnection for which the value(s) will be set

[in] path

Path to the Node(s) for which the value(s) will be set

[in/
out] wstr

Pointer to a null terminated unicode string (max 64k characters)

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_READONLY on attempt to set a read-only node
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIGetValueStringUnicode](#), [ziAPISetValueStringUnicode](#)

ziAPISync

ZIResult_enum ziAPISync (**ZIConnection** conn)

Synchronizes the session by dropping all pending data.

This function drops any data that is pending for transfer. Any data (including poll data) retrieved afterwards is guaranteed to be produced not earlier than the call to ziAPISync. This ensures in particular that any settings made prior to the call to ziAPISync have been propagated to the device, and the data retrieved afterwards is produced with the new settings already set to the hardware. Note, however, that this does not include any required settling time.

Parameters:

[in] conn

Pointer to the ZIConnection that is to be synchronized

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIEchoDevice

ZIResult_enum ziAPIEchoDevice (**ZIConnection** conn, const char* deviceSerial)

Sends an echo command to a device and blocks until answer is received.

This is useful to flush all buffers between API and device to enforce that further code is only executed after the device executed a previous command. Per device echo is only implemented for HF2. For other device types it is a synonym to ziAPISync, and deviceSerial parameter is ignored.

Parameters:

[in] conn

Pointer to the ZIConnection that is to be synchronized

[in] deviceSerial

The serial of the device to get the echo from, e.g., dev2100

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIAllocateEventEx

ZIEvent* ziAPIAllocateEventEx ()

Allocates [ZIEvent](#) structure and returns the pointer to it. Attention!!! It is the client code responsibility to deallocate the structure by calling [ziAPIDeallocateEventEx](#)!

This function allocates a [ZIEvent](#) structure and returns the pointer to it. Free the memory using [ziAPIDeallocateEventEx](#).

See Also:

[ziAPIDeallocateEventEx](#)

ziAPIDeallocateEventEx

void ziAPIDeallocateEventEx ([ZIEvent*](#) ev)

Deallocates [ZIEvent](#) structure created with [ziAPIAllocateEventEx\(\)](#).

Parameters:

[in] ev
Pointer to [ZIEvent](#) structure to be deallocated..

See Also:

[ziAPIAllocateEventEx](#)

This function is the compliment to [ziAPIAllocateEventEx\(\)](#)

ziAPISubscribe

ZIResult_enum ziAPISubscribe (**ZIConnection** conn, const char* path)

subscribes the nodes given by path for [ziAPIPollDataEx](#)

This function subscribes to nodes so that whenever the value of the node changes the new value can be polled using [ziAPIPollDataEx](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one leaf can be subscribed to with one function call.

Parameters:

[in] conn

Pointer to the ziConnection for which to subscribe for

[in] path

Path to the nodes to subscribe

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPIUnSubscribe](#), [ziAPIPollDataEx](#), [ziAPIGetValueAsPollData](#)

ziAPIUnSubscribe

ZIResult_enum ziAPIUnSubscribe (**ZIConnection** conn, const char* path)

unsubscribes to the nodes given by path

This function is the complement to [ziAPISubscribe](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one node can be unsubscribed with one function call.

Parameters:

[in] conn

Pointer to the ziConnection for which to unsubscribe for

[in] path

Path to the Nodes to unsubscribe

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no node given by path is able to hold values
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIPollDataEx](#), [ziAPIGetValueAsPollData](#)

ziAPIPollDataEx

ZIResult_enum ziAPIPollDataEx (**ZIConnection** conn, **ZIEvent*** ev, uint32_t timeoutMilliseconds)

checks if an event is available to read

This function returns immediately if an event is pending. Otherwise it waits for an event for up to timeoutMilliseconds. All value changes that occur in nodes that have been subscribed to or in children of nodes that have been subscribed to are sent from the Data Server to the ziAPI session. For a description of how the data are available in the struct, refer to the documentation of struct [ziEvent](#). When no event was available within timeoutMilliseconds, the ziEvent::Type field will be ZI_DATA_NONE and the ziEvent::Count field will be zero. Otherwise these fields hold the values corresponding to the event that occurred.

Parameters:

[in] conn

Pointer to the [ZIConnection](#) for which events should be received

[out] ev

Pointer to a [ZIEvent](#) struct in which the received event will be written

[in] timeoutMilliseconds

Time to wait for an event in milliseconds. If -1 it will wait forever, if 0 the function returns immediately.

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIGetValueAsPollData](#), [ziEvent](#)

ziAPIGetValueAsPollData

ZIResult_enum ziAPIGetValueAsPollData ([ZIConnection](#) conn, const char* path)

triggers a value request, which will be given back on the poll event queue

Use this function to receive the value of one or more nodes as one or more events using [ziAPIPollDataEx](#), even when the node is not subscribed or no value change has occurred.

Parameters:

[in] conn

Pointer to the [ZIConnection](#) with which the value should be retrieved

[in] path

Path to the Node holding the value

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the Path's Length exceeds MAX_PATH_LEN or the length of the char-buffer for the nodes given by MaxLen is too small for all elements
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in the Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIPollDataEx](#)

ziAPIAsyncSetDoubleData

ZIResult_enum ziAPIAsyncSetDoubleData (**ZIConnection** conn, const char* path, ZIDoubleData value)

ziAPIAsyncSetIntegerData

ZIResult_enum ziAPIAsyncSetIntegerData (**ZIConnection** conn, const char* path, ZIIntegerData value)

ziAPIAsyncSetByteArray

ZIResult_enum ziAPIAsyncSetByteArray (**ZIConnection** conn, const char* path, uint8_t* buffer, uint32_t length)

ziAPIAsyncSetString

ZIResult_enum ziAPIAsyncSetString (**ZIConnection** conn, const char* path, const char* str)

ziAPIAsyncSetStringUnicode

ZIResult_enum ziAPIAsyncSetStringUnicode (**ZIConnection** conn, const char* path, const wchar_t* wstr)

ziAPIAsyncSubscribe

[ZIResult_enum](#) ziAPIAsyncSubscribe ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

ziAPIAsyncUnSubscribe

[ZIResult_enum](#) ziAPIAsyncUnSubscribe ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

ziAPIAsyncGetValueAsPollData

[ZIResult_enum](#) ziAPIAsyncGetValueAsPollData ([ZIConnection](#) conn, const char* path, ZIAsyncTag tag)

ziAPIGetError

ZIResult_enum ziAPIGetError (**ZIResult_enum** result, char** buffer, int* base)

Returns a description and the severity for a **ZIResult_enum**.

This function returns a static char pointer to a description string for the given **ZIResult_enum** error code. It also provides a parameter returning the severity (info, warning, error). If the given error code does not exist a description for an unknown error and the base for an error will be returned. If a description or the base is not needed NULL may be passed. In general, it's recommended to use **ziAPIGetLastError** instead to get detailed error messages.

Parameters:

[in] result

A **ZIResult_enum** for which the description or base will be returned

[out] buffer

A pointer to a char array to return the description. May be NULL if no description is needed.

[out] base

The severity for the provided Status parameter:

- ZI_INFO_BASE For infos.
- ZI_WARNING_BASE For warnings.
- ZI_ERROR_BASE For errors.

Returns:

- ZI_INFO_SUCCESS Upon success.

ziAPIGetLastError

ZIResult_enum ziAPIGetLastError (**ZIConnection** conn, char* buffer, uint32_t bufferSize)

Returns the message from the last error that occurred.

This function can be used to obtain the error message from the last error that occurred associated with the provided **ZIConnection**. If the last ziAPI call is successful, then the last error message returned by ziAPIGetError is empty. Only ziAPI function calls that take **ZIConnection** as an input argument influence the message returned by ziAPIGetLastError, if they do not take **ZIConnection** as an input argument the last error message will neither be reset to be empty or set to an error message (in the case of the error). There are some exceptions to this rule, ziAPIGetLastError can also not be used with **ziAPIInit**, **ziAPIConnect**, **ziAPIConnectEx** and **ziAPIDestroy**. Note, a call to ziAPIGetLastError will also reset the last error message to empty if its call was successful. Since the buffer is left unchanged in the case of an error occurring in the call to ziAPIGetLastError it is safest to initialize the buffer with a known value, for example, "ziAPIGetLastError was not successful".

Parameters:

[in] conn

The **ZIConnection** from which to get the error message.

[out] buffer

A pointer to a char array to return the message.

[in] bufferSize

The length of the provided buffer.

Returns:

- **ZI_INFO_SUCCESS** Upon success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred. In this case the provided buffer is left unchanged.
- **ZI_ERROR_LENGTH** If the message's length exceeds the provided bufferSize, the message is truncated and written to buffer.

ziAPISetDebugLevel

void ziAPISetDebugLevel (int32_t debugLevel)

Enable ziAPI's log and set the severity level of entries to be included in the log.

Calling this function enables ziAPI's log at the specified severity level. On Windows the logs can be found by navigating to the Zurich Instruments "Logs" folder entry in the Windows Start Menu: Programs -> Zurich Instruments -> LabOne Servers -> Logs. This will open an Explorer window displaying folders containing log files from various LabOne components, in particular, the `ziAPILog` folder contains logs from ziAPI. On Linux, the logs can be found at `/tmp/ziAPILog_USERNAME`, where "USERNAME" is the same as the output of the "whoami" command.

Parameters:

[in] debugLevel

An integer specifying the log's severity level:

- trace: 0,
- info: 1,
- debug: 2,
- warning: 3,
- error: 4,
- fatal: 5,
- status: 6.

See Also:

[ziAPIWriteDebugLog](#)

ziAPIWriteDebugLog

void ziAPIWriteDebugLog (int32_t debugLevel, const char* message)

Write a message to ziAPI's log with the specified severity.

This function may be used to write a message to ziAPI's log from client code to assist with debugging. Note, this function is only available if the implementation used in [ziAPIConnectEx](#) is "ziAPI_Core" (the default implementation). Also logging must be first enabled using [ziAPISetDebugLevel](#).

Parameters:

[in] debugLevel

An integer specifying the severity of the message to write in the log:

- trace: 0,
- info: 1,
- debug: 2,
- warning: 3,
- error: 4,
- fatal: 5,
- status: 6.

[in] message

A character array comprising of the message to be written.

See Also:

[ziAPISetDebugLevel](#)

ReadMEMFile

[ZIResult_enum](#) ReadMEMFile (const char* filename, char* buffer, int32_t bufferSize, int32_t* bytesUsed)

ziAPIModCreate

ZIResult_enum ziAPIModCreate (**ZIConnection** conn, **ZIModuleHandle*** handle, const char* moduleId)

Create a **ZIModuleHandle** that can be used for asynchronous measurement tasks.

This function initializes a ziCore module and provides a pointer (handle) with which to access and work with it. Note that this function does not start the module's thread. Before the thread can be started (with **ziAPIModExecute**):

- the device serial (e.g., "dev100") to be used with module must be specified via **ziAPIModSetByteArray**.
- the desired data (node paths) to record during the measurement must be specified via **ziAPIModSubscribe**. The module's thread is stopped with **ziAPIModClear**.

Parameters:

[in] conn

The **ZIConnection** which should be used to initialize the module.

[out] handle

Pointer to the initialized **ZIModuleHandle**, which from then on can be used to reference the module.

[in] moduleId

The name specifying the type the module to create (only the following ziCore Modules are currently supported in ziAPI):

- "sweep" to initialize an instance of the Sweeper Module.
- "record" to initialize an instance of the Software Trigger (Recorder) Module.
- "zoomFFT" to initialize an instance of the Spectrum Module.
- "deviceSettings" to initialize an instance to save/load device settings.
- "pidAdvisor" to initialize an instance of the PID Advisor Module.
- "awgModule" to initialize an instance of the AWG Compiler Module.
- "impedanceModule" to initialize an instance of the Impedance Compensation Module.
- "scopeModule" to initialize an instance of the Scope Module to assembly scope shots.
- "multiDeviceSyncModule" to initialize an instance of the Device Synchronization Module.
- "dataAcquisitionModule" to initialize an instance of the Data Acquisition Module.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred.

- ZI_WARNING_NOTFOUND if the provided moduleId was invalid.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModClear](#)

ziAPIModSetDoubleData

ZIResult_enum ziAPIModSetDoubleData (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIDoubleData** value)

Sets a module parameter to the specified double type.

This function is used to configure (set) module parameters which have double types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] value

The double data to write to the path.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#), [ziAPIModSetString](#)

ziAPIModSetIntegerData

ZIResult_enum ziAPIModSetIntegerData (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIntegerData** value)

Sets a module parameter to the specified integer type.

This function is used to configure (set) module parameters which have integer types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] value

The integer data to write to the path.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetByteArray](#), [ziAPIModSetString](#)

ziAPIModSetByteArray

ZIResult_enum ziAPIModSetByteArray ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* path, uint8_t* buffer, uint32_t length)

Sets a module parameter to the specified byte array.

This function is used to configure (set) module parameters which have byte array types.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] buffer

Pointer to the byte array with the data.

[in] length

Length of the data in the buffer.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetString](#)

ziAPIModSetString

ZIResult_enum ziAPIModSetString (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, const char* str)

Sets a module parameter to the specified null-terminated string.

This function is used to configure (set) module parameters which have string types.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] str

Pointer to a null-terminated string (max 64k characters).

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#)

ziAPIModSetStringUnicode

ZIResult_enum ziAPIModSetStringUnicode ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* path, const wchar_t* wstr)

Sets a module parameter to the specified null-terminated unicode string.

This function is used to configure (set) module parameters which have string types.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to set data on.

[in] path

Path to the module parameter path.

[in] wstr

Pointer to a null-terminated unicode string (max 64k characters).

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSetDoubleData](#), [ziAPIModSetIntegerData](#), [ziAPIModSetByteArray](#)

ziAPIModGetInteger

ZIResult_enum ziAPIModGetInteger (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIntegerData*** value)

gets the integer-type value of the specified node

This function is used to retrieve module parameter values of type integer.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the node holding the value

[out] value

Pointer to an 64bit integer in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetDouble](#), [ziApiModGetString](#)

ziAPIModGetDouble

ZIResult_enum ziAPIModGetDouble (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, **ZIDoubleData*** value)

gets the double-type value of the specified node

This function is used to retrieve module parameter values of type floating point double.

Parameters:

[in] conn

Pointer to ZIConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the node holding the value

[out] value

Pointer to an floating point double in which the value should be written

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetString](#)

ziAPIModGetString

ZIResult_enum ziAPIModGetString (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, char* buffer, unsigned int* length, unsigned int bufferSize)

gets the null-terminated string value of the specified node

This function is used to retrieve module parameter values of type string.

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the Node holding the value

[out] buffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetDouble](#)

ziAPIModGetStringUnicode

ZIResult_enum ziAPIModGetStringUnicode (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, wchar_t* wbuffer, unsigned int* length, unsigned int bufferSize)

gets the null-terminated string value of the specified node

This function is used to retrieve module parameter values of type string.

Parameters:

[in] conn

Pointer to the ziConnection with which the value should be retrieved

[in] handle

The **ZIModuleHandle** specifying the module in which the nodes should be subscribed to.

[in] path

Path to the Node holding the value

[out] wbuffer

Pointer to a buffer to store the retrieved null-terminated string

[out] length

Pointer to an unsigned int to store the length of the string in (including the null terminator). If an error occurred or the length of the passed buffer is insufficient, a zero will be returned

[in] bufferSize

The length of the passed buffer

Returns:

- ZI_INFO_SUCCESS on success
- ZI_ERROR_CONNECTION when the connection is invalid (not connected) or when a communication error occurred
- ZI_ERROR_LENGTH if the path's length exceeds MAX_PATH_LEN
- ZI_WARNING_OVERFLOW when a FIFO overflow occurred
- ZI_ERROR_COMMAND on an incorrect answer of the server
- ZI_ERROR_SERVER_INTERNAL if an internal error occurred in Data Server
- ZI_WARNING_NOTFOUND if the given path could not be resolved or no value is attached to the node
- ZI_ERROR_TIMEOUT when communication timed out
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetInteger](#), [ziApiModGetDouble](#), [ziAPIModGetString](#)

ziAPIModListNodes

ZIResult_enum ziAPIModListNodes (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path, char* nodes, uint32_t bufferSize, uint32_t flags)

Returns all child parameter node paths found under the specified parent module parameter path.

This function returns a list of parameter names found at the specified path. The path may contain wildcards. The list is returned in a null-terminated char-buffer, each element delimited by a newline. If the maximum length of the buffer (bufferSize) is not sufficient for all elements, nothing will be returned and the return value will be **ZI_ERROR_LENGTH**. Note, the provided path must match the module being addressed, i.e., path must exactly start with "sweep/" for the Sweeper Module.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** from which the parameter names should be retrieved.

[in] path

Path for which all children will be returned. The path may contain wildcard characters.

[out] nodes

Upon call filled with newline-delimited list of the names of all the children found. The string is zero-terminated.

[in] bufferSize

The length of the buffer specified as the nodes output parameter.

[in] flags

A combination of flags (applied bitwise) as defined in **ZIListNodes_enum**.

Returns:

- **ZI_INFO_SUCCESS** On success
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_LENGTH** If the path's length exceeds **MAX_PATH_LEN** or the length of the char-buffer for the nodes given by bufferSize is too small for all elements.
- **ZI_WARNING_OVERFLOW** When a FIFO overflow occurred.
- **ZI_ERROR_COMMAND** On an incorrect answer of the server.
- **ZI_ERROR_SERVER_INTERNAL** If an internal error occurred in Data Server.
- **ZI_WARNING_NOTFOUND** If the given path could not be resolved.
- **ZI_ERROR_TIMEOUT** When communication timed out.

- `ZI_ERROR_GENERAL` If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIModSubscribe

ZIResult_enum ziAPIModSubscribe (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path)

Subscribes to the nodes specified by path, these nodes will be recorded during module execution.

This function subscribes to nodes so that whenever the value of the node changes while the module is executing the new value will be accumulated and then read using [ziAPIModRead](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one leaf can be subscribed to with one function call.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module in which the nodes should be subscribed to.

[in] path

Path specifying the nodes to subscribe to, may contain wildcards.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or a general error occurred, enable ziAPI's log for detailed information, see [ziAPISetDebugLevel](#).
- ZI_ERROR_LENGTH If the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW When a FIFO overflow occurred.
- ZI_ERROR_COMMAND On an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL If an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND If the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT When communication timed out.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModUnSubscribe](#), [ziAPIModRead](#)

ziAPIModUnSubscribe

ZIResult_enum ziAPIModUnSubscribe (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path)

Unsubscribes to the nodes specified by path.

This function is the complement to [ziAPIModSubscribe](#). By using wildcards or by using a path that is not a leaf node but contains sub nodes, more than one node can be unsubscribed with one function call.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module in which the nodes should be unsubscribed from.

[in] path

Path specifying the nodes to unsubscribe from, may contain wildcards.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_LENGTH If the Path's Length exceeds MAX_PATH_LEN.
- ZI_WARNING_OVERFLOW When a FIFO overflow occurred.
- ZI_ERROR_COMMAND On an incorrect answer of the server.
- ZI_ERROR_SERVER_INTERNAL If an internal error occurred in the Data Server.
- ZI_WARNING_NOTFOUND If the given path could not be resolved or no node given by path is able to hold values.
- ZI_ERROR_TIMEOUT When communication timed out.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModSubscribe](#), [ziAPIModRead](#)

ziAPIModExecute

ZIResult_enum [ziAPIModExecute](#) ([ZIConnection](#) conn, [ZIModuleHandle](#) handle)

Starts the module's thread and its associated measurement task.

Once the module's parameters has been configured as required via, e.g. [ziAPIModSetDoubleData](#), this function starts the module's thread. This starts the module's main measurement task which will run asynchronously. The thread will run until either the module has completed its task or until [ziAPIModFinish](#) is called. Subscription or unsubscription is not possible while the module is executing. The status of the module can be obtained with either [ziAPIModFinished](#) or [ziAPIModProgress](#).

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModCreate](#), [ziAPIModProgress](#), [ziAPIModFinish](#)

ziAPIModTrigger

ZIResult_enum ziAPIModTrigger ([ZIConnection](#) conn, [ZIModuleHandle](#) handle)

Manually issue a trigger forcing data recording (SW Trigger Module only).

This function is used with the Software Trigger Module in order to manually issue a trigger in order to force recording of data. A burst of subscribed data will be recorded as configured via the SW Trigger's parameters as would a regular trigger event.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

ziAPIModProgress

ZIResult_enum ziAPIModProgress (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIDoubleData*** progress)

Queries the current state of progress of the module's measurement task.

This function can be used to query the module's progress in performing its current measurement task, the progress is returned as a double in [0, 1], where 1 indicates task completion.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] progress

A pointer to **ZIDoubleData** indicating the current progress of the module.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

ziAPIModExecute, **ziAPIModFinish**, **ziAPIModFinished**

ziAPIModFinished

ZIResult_enum ziAPIModFinished (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIIntegerData*** finished)

Queries whether the module has finished its measurement task.

This function can be used to query whether the module has finished its task or not.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] finished

A pointer to **ZIIntegerData**, upon return this will be 0 if the module is still executing or 1 if it has finished executing.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#), [ziAPIModProgress](#)

ziAPIModFinish

ZIResult_enum ziAPIModFinish (**ZIConnection** conn, **ZIModuleHandle** handle)

Stops the module performing its measurement task.

This functions stops the module performing its associated measurement task and stops recording any data. The task and data recording may be restarted by calling [ziAPIModExecute](#) again.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModProgress](#), [ziAPIModFinished](#)

ziAPIModSave

ZIResult_enum ziAPIModSave ([ZIConnection](#) conn, [ZIModuleHandle](#) handle, const char* fileName)

Saves the currently accumulated data to file.

This function saves the currently accumulated data to a file. The path of the file to save data to is specified via the module's directory parameter.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[in] fileName

The basename of the file to save the data in.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#), [ziAPIModFinished](#)

ziAPIModRead

ZIResult_enum ziAPIModRead (**ZIConnection** conn, **ZIModuleHandle** handle, const char* path)

Make the currently accumulated data available for use in the C program.

This function can be used to either read (get) module parameters, in this case a path that addresses the module must be specified, or it can be used to read out the currently accumulated data from subscribed nodes in the module. In either case the actual data must then be accessed by the user using [ziAPIModNextNode](#) and [ziAPIModGetChunk](#).

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[in] path

The path specifying the module parameter(s) to get, specify NULL to obtain all subscribed data.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModGetChunk](#), [ziAPIModNextNode](#)

ziAPIModNextNode

ZIResult_enum ziAPIModNextNode (**ZIConnection** conn, **ZIModuleHandle** handle, char* path, uint32_t bufferSize, **ZIValueType_enum*** valueType, uint64_t* chunks)

Make the data for the next node available for reading with [ziAPIModGetChunk](#).

After calling [ziAPIModRead](#), subscribed data (or module parameters) may now be read out on a node-by-node and chunk-by-chunk basis. All nodes with data available in the module can be iterated over by using [ziAPIModNextNode](#), then for each node the chunks of data available are read out using [ziAPIModGetChunk](#). Calling this function makes the data from the next node available for read.

Parameters:

[in] conn

The [ZIConnection](#) from which the module was created.

[in] handle

The [ZIModuleHandle](#) specifying the module to execute.

[out] path

A string specifying the node's path whose data chunk points to.

[in] bufferSize

The length of the buffer specified as the path output parameter.

[out] valueType

The [ZIValueType_enum](#) of the node's data.

[out] chunks

The number of chunks of data available for the node.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModRead](#), [ziAPIModGetChunk](#), [ziAPIModEventDeallocate](#)

ziAPIModGetChunk

ZIResult_enum ziAPIModGetChunk (**ZIConnection** conn, **ZIModuleHandle** handle, **uint64_t** chunkIndex, **ZIModuleEventPtr*** ev)

Get the specified data chunk from the current node.

Data is read out node-by-node and then chunk-by-chunk. This function can be used to obtain specific data chunks from the current node that data is being read from. More precisely, it preallocates space for an event structure big enough to hold the node's data at the specified chunk index, updates **ZIModuleEventPtr** to point to this space and then copies the chunk data to this space.

Note, before the very first call to ziAPIModGetChunk, the **ZIModuleEventPtr** should be initialized to NULL and then left untouched for all subsequent calls (even after calling **ziAPIModNextNode** to get data from the next node). This is because ziAPIModGetChunk internally manages the required space allocation for the event and then in subsequent calls only reallocates space when it is required. It is optimized to reduce the number of required space reallocations for the event.

The **ZIModuleEventPtr** should be deallocated using **ziAPIModEventDeallocate**, otherwise the lifetime of the **ZIModuleEventPtr** is the same as the lifetime of the module. Indeed, the same **ZIModuleEventPtr** can be used, even for subsequent reads. It is also possible to work with multiple **ZIModuleEventPtr** so that some pointers can be kept for later processing.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[out] chunkIndex

The index of the data chunk to update the pointer to.

[out] ev

The module's **ZIModuleEventPtr** that points to the currently available data chunk.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

[ziAPIModRead](#), [ziAPIModNextNode](#), [ziAPIModEventDeallocate](#)

ziAPIModEventDeallocate

ZIResult_enum ziAPIModEventDeallocate (**ZIConnection** conn, **ZIModuleHandle** handle, **ZIModuleEventPtr** ev)

Deallocate the **ZIModuleEventPtr** being used by the module.

This function deallocates the **ZIModuleEventPtr**. Since a module event's allocated space is managed internally by **ziAPIModGetChunk**, when the user no longer requires the event (all data has been read out) it must be deallocated by the user with this function.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

[in] ev

The **ZIModuleEventPtr** to deallocate.

Returns:

- **ZI_INFO_SUCCESS** On success.
- **ZI_ERROR_CONNECTION** When the connection is invalid (not connected) or when a communication error occurred.
- **ZI_ERROR_GENERAL** If a general error occurred, use **ziAPIGetLastError** for a detailed error message.
- Other return codes may also be returned, for a detailed error message use **ziAPIGetLastError**.

See Also:

ziAPIModGetChunk, **ziAPIModRead**

ziAPIModClear

ZIResult_enum ziAPIModClear (**ZIConnection** conn, **ZIModuleHandle** handle)

Terminates the module's thread and destroys the module.

This function terminates the module's thread. After calling ziAPIModClear the module's handle may not be used any more. A new instance of the module must be initialized if required.

Parameters:

[in] conn

The **ZIConnection** from which the module was created.

[in] handle

The **ZIModuleHandle** specifying the module to execute.

Returns:

- ZI_INFO_SUCCESS On success.
- ZI_ERROR_CONNECTION When the connection is invalid (not connected) or when a communication error occurred.
- ZI_ERROR_GENERAL If a general error occurred, use [ziAPIGetLastError](#) for a detailed error message.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIModExecute](#), [ziAPIModFinish](#)

ziAPIVectorWriteBlock

ZIResult_enum ziAPIVectorWriteBlock (**ZIConnection** conn, const char* path, **ZIVectorData*** vectorBlock)

ziAPIVectorWriteGetStatus

ZIResult_enum ziAPIVectorWriteGetStatus (**ZIConnection** conn, const char* path, uint8_t* status)

status - see ::ZIVectorWriteStatus_enum

ziAPIVectorWrite

[ZIResult_enum](#) ziAPIVectorWrite ([ZIConnection](#) conn, const char* path, const void* vectorPtr, uint8_t vectorElementType, uint64_t vectorSizeElements)

vectorElementType - see [ZIVectorElementType_enum](#)

ziAPIDiscoveryFindAll

ZIResult_enum ziAPIDiscoveryFindAll (**ZIConnection** conn, char* deviceIds, uint32_t bufferSize)

Perform a Discovery property look-up for the specified deviceAddress and return its device ID. Attention! This invalidates all pointers previously returned by ziAPIDiscovery* calls. The deviceId need not be deallocated by the user.

Parameters:

[in] conn

Pointer to **ZIConnection** with which the value should be retrieved.

[out] deviceIds

Pointer to a buffer that is to contain the list of newline-separated IDs of the devices found, e.g. "DEV2006\nDEV2007\n".

[in] bufferSize

The size of the buffer pointed to by deviceIds. If the buffer is too small to hold the complete list of device IDs, its contents remain unchanged.

Returns:

- ZI_INFO_SUCCESS
- ZI_ERROR_LENGTH The provided buffer is too small to hold the list.
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryFind

ZIResult_enum ziAPIDiscoveryFind (**ZIConnection** conn, const char* deviceAddress, const char** deviceId)

Perform a Discovery property look-up for the specified deviceAddress and return its device ID. Attention! This invalidates all pointers previously returned by ziAPIDiscovery* calls. The deviceId need not be deallocated by the user.

Parameters:

[in] conn

Pointer to **ZIConnection** with which the value should be retrieved.

[in] deviceAddress

The address or ID of the device to find, e.g., 'uhf-dev2006' or 'dev2006'.

[out] deviceId

The ID of the device that was found, e.g. 'DEV2006'.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFindAll](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGet

ZIResult_enum ziAPIDiscoveryGet (**ZIConnection** conn, const char* deviceId, const char** propsJSON)

Returns the device Discovery properties for a given device ID in JSON format. The function [ziAPIDiscoveryFind](#) must be called before ziAPIDiscoveryGet can be used. The propsJSON need not be deallocated by the user.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[out] propsJSON

The Discovery properites in JSON format of the specified device.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGetValueI](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGetValueI

ZIResult_enum ziAPIDiscoveryGetValueI (**ZIConnection** conn, const char* deviceId, const char* propName, ZIIntegerData* value)

Returns the specified integer Discovery property value for a given device ID. The function [ziAPIDiscoveryFind](#) must be called with the required device ID before using [ziAPIDiscoveryGetValueI](#).

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[in] propName

The name of the desired integer Discovery property.

[out] value

Pointer to the value of the specified Discovery property.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueS](#)

ziAPIDiscoveryGetValueS

ZIResult_enum ziAPIDiscoveryGetValueS (**ZIConnection** conn, const char* deviceId, const char* propName, const char** value)

Returns the specified string Discovery property value for a given device ID. The function [ziAPIDiscoveryFind](#) must be called with the required device ID before using ziAPIDiscoveryGetValueS. The value must not be deallocated by the user.

Parameters:

[in] conn

Pointer to [ZIConnection](#) with which the value should be retrieved.

[in] deviceId

The ID of the device to get Discovery information for, as returned by [ziAPIDiscoveryFind](#), e.g., 'dev2006'.

[in] propName

The name of the desired integer Discovery property.

[out] value

Pointer to the value of the specified Discovery property.

Returns:

- ZI_INFO_SUCCESS
- Other return codes may also be returned, for a detailed error message use [ziAPIGetLastError](#).

See Also:

[ziAPIDiscoveryFind](#), [ziAPIDiscoveryGet](#), [ziAPIDiscoveryGetValueI](#)

ziAPIAllocateEvent

`__inline ziEvent* ziAPIAllocateEvent ( )`

Deprecated: See [ziAPIAllocateEventEx\(\)](#).

ziAPIDeallocateEvent

```
__inline void ziAPIDeallocateEvent ( ziEvent* ev )
```

Deprecated: See [ziAPIDeallocateEventEx\(\)](#).

ziAPIPollData

`__inline ZIResult\_enum ziAPIPollData ( ZIConnection conn, ziEvent\* ev, int timeOut )`

Checks if an event is available to read. Deprecated: See [ziAPIPollDataEx\(\)](#).

Parameters:

[in] conn

Pointer to the [ZIConnection](#) for which events should be received

[out] ev

Pointer to a [ziEvent](#) struct in which the received event will be written

[in] timeOut

Time to wait for an event in milliseconds. If -1 it will wait forever, if 0 the function returns immediately.

Returns:

- [ZI_SUCCESS](#) On success.
- [ZI_CONNECTION](#) When the connection is invalid (not connected) or when a communication error occurred.
- [ZI_OVERFLOW](#) When a FIFO overflow occurred.

See [Data Handling](#) for an example

See Also:

[ziAPISubscribe](#), [ziAPIUnSubscribe](#), [ziAPIGetValueAsPollData](#), [ziEvent](#)

ziAPIGetValueS

```
__inline ZIResult_enum ziAPIGetValueS ( ZIConnection conn, char* path,  
DemodSample* value )
```


ziAPIGetValueDIO

```
__inline ZIResult_enum ziAPIGetValueDIO ( ZIConnection conn, char* path,  
DIOSample* value )
```

ziAPIGetValueAuxIn

```
__inline ZIResult_enum ziAPIGetValueAuxIn ( ZIConnection conn, char* path,  
AuxInSample* value )
```

ziAPISecondsTimeStamp

double ziAPISecondsTimeStamp (ziTimeStampType TS)

Deprecated: timestamps should instead be converted to seconds by dividing by the instrument's "clockbase". This is available as an leaf under the instrument's root "device" branch in the node hierarchy, e.g., /dev2001/clockbase.

Parameters:

[in] TS
the timestamp to convert to seconds

Returns:

The timestamp in seconds as a double

Glossary

This glossary provides easy to understand descriptions for many terms related to measurement instrumentation including the abbreviations used inside this user manual.

A

A/D	Analog to Digital See Also ADC .
AC	Alternate Current
ADC	Analog to Digital Converter
AM	Amplitude Modulation
Amplitude Modulated AFM (AM-AFM)	AFM mode where the amplitude change between drive and measured signal encodes the topography or the measured AFM variable. See Also Atomic Force Microscope .
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
Atomic Force Microscope (AFM)	Microscope that scans surfaces by means an oscillating mechanical structure (e.g. cantilever, tuning fork) whose oscillating tip gets so close to the surface to enter in interaction because of electrostatic, chemical, magnetic or other forces. With an AFM it is possible to produce images with atomic resolution. See Also Amplitude Modulated AFM , Frequency Modulated AFM , Phase modulation AFM .
AVAR	Allen Variance

B

Bandwidth (BW)	The signal bandwidth represents the highest frequency components of interest in a signal. For filters the signal bandwidth is the cut-off point, where the transfer function of a system shows 3 dB attenuation versus DC. In this context the bandwidth is a synonym of cut-off frequency $f_{\text{cut-off}}$ or 3dB frequency $f_{-3\text{dB}}$. The concept of bandwidth is used when the dynamic behavior of a signal is important or separation of different signals is required. In the context of a open-loop or closed-loop system, the bandwidth can be used to indicate the fastest speed of the system, or the highest signal update change rate that is possible with the system. Sometimes the term bandwidth is erroneously used as synonym of frequency range. See Also Noise Equivalent Power Bandwidth.
BNC	Bayonet Neill-Concelman Connector

C

CF	Clock Fail (internal processor clock missing)

Common Mode Rejection Ratio (CMRR)	Specification of a differential amplifier (or other device) indicating the ability of an amplifier to obtain the difference between two inputs while rejecting the components that do not differ from the signal (common mode). A high CMRR is important in applications where the signal of interest is represented by a small voltage fluctuation superimposed on a (possibly large) voltage offset, or when relevant information is contained in the voltage difference between two signals. The simplest mathematical definition of common-mode rejection ratio is: $CMRR = 20 * \log(\text{differential gain} / \text{common mode gain})$.

CSV	Comma Separated Values

D

D/A	Digital to Analog

DAC	Digital to Analog Converter

DC	Direct Current

DDS	Direct Digital Synthesis

DHCP	Dynamic Host Configuration Protocol

DIO	Digital Input/Output

DNS	Domain Name Server

DSP	Digital Signal Processor

DUT	Device Under Test

Dynamic Reserve (DR)	The measure of a lock-in amplifier's capability to withstand the disturbing signals and noise at non-reference frequencies, while maintaining the specified measurement accuracy within the signal bandwidth.

E

XML	Extensible Markup Language. See Also XML .

F

FFT	Fast Fourier Transform

FIFO	First In First Out

FM	Frequency Modulation

Frequency Accuracy (FA)	Measure of an instrument's ability to faithfully indicate the correct frequency versus a traceable standard.

Frequency Modulated AFM (FM-AFM)	AFM mode where the frequency change between drive and measured signal encodes the topography or the measured AFM variable. See Also Atomic Force Microscope .

Frequency Response Analyzer (FRA)	Instrument capable to stimulate a device under test and plot the frequency response over a selectable frequency range with a fine granularity.

Frequency Sweeper

See Also [Frequency Response Analyzer](#).

G

Gain Phase Meter

See Also [Vector Network Analyzer](#).

GPIO

General Purpose Interface Bus

GUI

Graphical User Interface

I

I/O

Input / Output

Impedance Spectroscope
(IS)

Instrument suited to stimulate a device under test and to measure the impedance (by means of a current measurement) at a selectable frequency and its amplitude and phase change over time. The output is both amplitude and phase information referred to the stimulus signal.

Input Amplitude Accuracy
(IAA)

Measure of instrument's capability to faithfully indicate the signal amplitude at the input channel versus a traceable standard.

Input voltage noise (IVN)

Total noise generated by the instrument and referred to the signal input, thus expressed as additional source of noise for the measured signal.

IP

Internet Protocol

L

LAN

Local Area Network

LED

Light Emitting Diode

Lock-in Amplifier (LI, LIA)

Instrument suited for the acquisition of small signals in noisy environments, or quickly changing signal with good signal to noise ratio - lock-in amplifiers recover the signal of interest knowing the frequency of the signal by demodulation with the suited reference frequency - the result of the demodulation are amplitude and phase of the signal compared to the reference: these are value pairs in the complex plane (X, Y) , (R, Θ) .

M

Media Access Control
address (MAC address)

Refers to the unique identifier assigned to network adapters for physical network communication.

Multi-frequency (MF)

Refers to the simultaneous measurement of signals modulated at arbitrary frequencies. The objective of multi-frequency is to increase the information that can be derived from a measurement which is particularly important for one-time, non-repeating events, and to increase the speed of a measurement since different frequencies do not have to be applied one after the other.

See Also [Multi-harmonic](#).

Multi-harmonic (MH)

Refers to the simultaneous measurement of modulated signals at various harmonic frequencies. The objective of multi-frequency is to increase the

information that can be derived from a measurement which is particularly important for one-time, non-repeating events, and to increase the speed of a measurement since different frequencies do not have to be applied one after the other.

See Also [Multi-frequency](#).

N

Noise Equivalent Power Bandwidth (NEPBW)

Effective bandwidth considering the area below the transfer function of a low-pass filter in the frequency spectrum. NEPBW is used when the amount of power within a certain bandwidth is important, such as noise measurements. This unit corresponds to a perfect filter with infinite steepness at the equivalent frequency.

See Also [Bandwidth](#).

Nyquist Frequency (NF)

For sampled analog signals, the Nyquist frequency corresponds to two times the highest frequency component that is being correctly represented after the signal conversion.

O

Output Amplitude Accuracy (OAA)

Measure of an instrument's ability to faithfully output a set voltage at a given frequency versus a traceable standard.

OV

Over Volt (signal input saturation and clipping of signal)

P

PC

Personal Computer

PD

Phase Detector

Phase-locked Loop (PLL)

Electronic circuit that serves to track and control a defined frequency. For this purpose a copy of the external signal is generated such that it is in phase with the original signal, but with usually better spectral characteristics. It can act as frequency stabilization, frequency multiplication, or as frequency recovery. In both analog and digital implementations it consists of a phase detector, a loop filter, a controller, and an oscillator.

Phase modulation AFM (PM-AFM)

AFM mode where the phase between drive and measured signal encodes the topography or the measured AFM variable.

See Also [Atomic Force Microscope](#).

PID

Proportional-Integral-Derivative

PL

Packet Loss (loss of packets of data between the instruments and the host computer)

R

RISC

Reduced Instruction Set Computer

Root Mean Square (RMS)

Statistical measure of the magnitude of a varying quantity. It is especially useful when variates are positive and negative, e.g., sinusoids, sawtooth, square waves. For a sine wave the following relation holds between the

amplitude and the RMS value: $U_{\text{RMS}} = U_{\text{PK}} / \sqrt{2} = U_{\text{PK}} / 1.41$. The RMS is also called quadratic mean.

RT Real-time

S

Scalar Network Analyzer (SNA) Instrument that measures the voltage of an analog input signal providing just the amplitude (gain) information.
See Also [Spectrum Analyzer](#), [Vector Network Analyzer](#).

SL Sample Loss (loss of samples between the instrument and the host computer)

Spectrum Analyzer (SA) Instrument that measures the voltage of an analog input signal providing just the amplitude (gain) information over a defined spectrum.
See Also [Scalar Network Analyzer](#).

SSH Secure Shell

T

TC Time Constant

TCP/IP Transmission Control Protocol / Internet Protocol

Thread An independent sequence of instructions to be executed by a processor.

Total Harmonic Distortion (THD) Measure of the non-linearity of signal channels (input and output)

TTL Transistor to Transistor Logic level

U

UHF Ultra-High Frequency

UHS Ultra-High Stability

USB Universal Serial Bus

V

VCO Voltage Controlled Oscillator

Vector Network Analyzer (VNA) Instrument that measures the network parameters of electrical networks, commonly expressed as s-parameters. For this purpose it measures the voltage of an input signal providing both amplitude (gain) and phase information. For this characteristic an older name was gain phase meter.
See Also [Gain Phase Meter](#), [Scalar Network Analyzer](#).

X

XML Extensible Markup Language: Markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

Z

ZCtrl	Zurich Instruments Control bus
ZoomFFT	This technique performs FFT processing on demodulated samples, for instance after a lock-in amplifier. Since the resolution of an FFT depends on the number of point acquired and the spanned time (not the sample rate), it is possible to obtain very highly resolution spectral analysis.
ZSync	Zurich Instruments Synchronization bus

Index

Symbols

.NET, 222-252
 Comparison to other interfaces, 10
 Examples, 228
 Getting started, 224
 Installing the API, 223
 Requirements, 223

A

API
 Compatibility, 13
 Levels, 13
 Versions, 13
Asynchronous commands, 31
AWG Module, 38-43

C

C API (see ziAPI)
C Programming Language (see ziAPI)
Comparison of LabOne APIs, 10
Compatibility
 Data Server and API, 16
 Software, 16

D

Data Acquisition Module, 44-58
Data Server, 8
 Node, 18
Data Streaming, 22
Device Settings Module, 59-59
Device Synchronisation Module, 62-62

I

Impedance Module, 60-61

L

LabOne
 API overview, 10
 Comparison of APIs, 10
LabOne Programming
 Quick Start Guide, 6-7
LabVIEW, 213-221
 Comparison to other interfaces, 10
 Concepts, 216
 Examples, finding, 218
 Examples, running, 218
 Finding examples, 218
 Finding help, 218
 Getting started, 216
 Installation, 214
 Linux, 215
 Mac, 215
 Windows, 214

 LabOne VI Palette, 216
 Modules, 217
 Palette, LabOne, 216
 Requirements, 214
 Running examples, 218
 Tips and tricks, 221
 VI Palette, 216
Low-level commands, 32-36

M

Matlab, 102-139
 Built-in help, 109
 Command reference, 116
 Comparison to other interfaces, 10
 Contents of the API package, 106
 Examples, running, 109
 Getting started, 106
 Help, accessing , 109
 Installation, 103, 103
 List of Examples, 106
 List of Utility functions, 106
 Logging, 111
 Modules, 110
 Modules, configuring, 110
 Reference, 116
 Requirements, 103
 Running examples, 109
 Tips and tricks, 112
 Troubleshooting, 114
 Verifying correct configuration, 104
MF
 Data Server, 28
Multi-Device Synchronisation Module, 62-62
Multi-threading, 13

N

Node
 Hierarchy, 18
 Leaf, 18
 Listing Nodes, 19
 Properties, 19
 Server node, 20
 Streaming nodes, 22
 Tree, 18
 Types, 19

P

PID Advisor Module, 63-70
PLL Module
 Deprecation notice, 66
Python, 140-212
 Built-in help, 144
 Command reference, 151
 AwgModule class, 170
 DataAcquisitionModule class, 173
 DeviceSettingsModule class, 180

- ImpedanceModule class, 183
- MultiDeviceSyncModule class, 186
- PidAdvisorModule class, 189
- RecorderModule class, 203
- ScopeModule class, 193
- SweeperModule class, 197
- zhinst package, 151
- zhinst's utility functions, 152
- ziDAQServer class, 161
- ziDiscovery class, 160
- ZoomFFTModule class, 208
- Comparison to other interfaces, 10
- Contents of the API package, 144
- Examples, common, 146
- Examples, HDAWG, 146
- Examples, HF2, 147
- Examples, running, 144
- Examples, UHF, 147
- Exploring the available examples, 145
- Getting started, 144
- Help, accessing , 144
- Installation, 142
- Installing the API, 141
- Loading data in Python, 150
- Locating the zhinst installation, 147
- Logging, 148
- Modules, 148
- Modules, configuring, 148
- Recommended python packages for ziPython, 142
- Reference, 151
- Requirements for using the Python API, 141
- Running examples, 144
- Tips and tricks, 150

Q

- Quick Start Guide
 - LabOne Programming, 6-7

S

- Scope Module, 71-78
- Software Trigger Module, 90-98
- Spectrum Module, 99-101
- Streaming, 22, 22
- Sweeper Module, 79-89
 - Bandwidth control, 79
 - Measurement data, 81
 - Measurement data, averaging, 81
 - Scanning mode, 79
 - Settling time, 80
 - Settling time, definition, 80
 - Sweep parameter, 79
 - Sweep range, 79
- Synchronous commands, 31

U

- UHF

- Automatic calibration, 28
- Calibration, 28

Z

- ziAPI
 - Comparison to other interfaces, 10
- ziAPI, C API functions and data types
 - MAX_EVENT_SIZE, 448
 - MAX_NAME_LEN, 448
 - MAX_PATH_LEN, 448
 - ReadMEMFile, 594
 - ziAPIAllocateEvent, 629
 - ziAPIAllocateEventEx, 319, 576
 - ziAPIAsyncGetValueAsPollData, 334, 589
 - ziAPIAsyncSetByteArray, 329, 584
 - ziAPIAsyncSetDoubleData, 327, 582
 - ziAPIAsyncSetIntegerData, 328, 583
 - ziAPIAsyncSetString, 330, 585
 - ziAPIAsyncSetStringUnicode, 331, 586
 - ziAPIAsyncSubscribe, 332, 587
 - ziAPIAsyncUnSubscribe, 333, 588
 - ziAPIConnect, 261, 535
 - ziAPIConnectDevice, 545
 - ziAPIConnectEx, 264, 538
 - ziAPIDeallocateEvent, 630
 - ziAPIDeallocateEventEx, 320, 577
 - ziAPIDestroy, 260, 534
 - ziAPIDisconnect, 262, 536
 - ziAPIDisconnectDevice, 546
 - ziAPIDiscoveryFind, 382, 625
 - ziAPIDiscoveryFindAll, 381, 624
 - ziAPIDiscoveryGet, 383, 626
 - ziAPIDiscoveryGetValueI, 384, 627
 - ziAPIDiscoveryGetValueS, 385, 628
 - ziAPIEchoDevice, 305, 575
 - ziAPIGetAuxInSample, 286, 556
 - ziAPIGetComplexData, 279, 549
 - ziAPIGetConnectionAPILevel, 265, 539
 - ziAPIGetDemodSample, 282, 552
 - ziAPIGetDIOSample, 284, 554
 - ziAPIGetError, 336, 590
 - ziAPIGetLastError, 337, 591
 - ziAPIGetRevision, 267, 541
 - ziAPIGetValueAsPollData, 324, 581
 - ziAPIGetValueAuxIn, 308, 634
 - ziAPIGetValueB, 288, 558
 - ziAPIGetValueD, 277, 547
 - ziAPIGetValueDIO, 307, 633
 - ziAPIGetValueI, 281, 551
 - ziAPIGetValueS, 306, 632
 - ziAPIGetValueString, 290, 560
 - ziAPIGetValueStringUnicode, 291, 561
 - ziAPIGetVersion, 266, 540
 - ziAPIInit, 259, 533
 - ziAPIListImplementations, 263, 537
 - ziAPIListNodes, 271, 542

ziAPIListNodesJSON, 273, 543
ziAPIModClear, 374, 620
ziAPIModCreate, 349, 595
ziAPIModEventDeallocate, 373, 619
ziAPIModExecute, 364, 610
ziAPIModFinish, 368, 614
ziAPIModFinished, 367, 613
ziAPIModGetChunk, 372, 618
ziAPIModGetDouble, 357, 603
ziAPIModGetInteger, 356, 602
ziAPIModGetString, 358, 604
ziAPIModGetStringUnicode, 359, 605
ziAPIModListNodes, 360, 606
ziAPIModNextNode, 371, 617
ziAPIModProgress, 366, 612
ziAPIModRead, 370, 616
ziAPIModSave, 369, 615
ziAPIModSetByteArray, 353, 599
ziAPIModSetDoubleData, 351, 597
ziAPIModSetIntegerData, 352, 598
ziAPIModSetString, 354, 600
ziAPIModSetStringUnicode, 355, 601
ziAPIModSubscribe, 362, 608
ziAPIModTrigger, 365, 611
ziAPIModUnSubscribe, 363, 609
ziAPIPollData, 325, 631
ziAPIPollDataEx, 323, 580
ziAPISecondsTimeStamp, 635
ziAPISetComplexData, 293, 563
ziAPISetDebugLevel, 338, 592
ziAPISetValueB, 295, 565
ziAPISetValueD, 292, 562
ziAPISetValueI, 294, 564
ziAPISetValueString, 297, 567
ziAPISetValueStringUnicode, 298, 568
ziAPISubscribe, 321, 578
ziAPISync, 304, 574
ziAPISyncSetValueB, 301, 571
ziAPISyncSetValueD, 299, 569
ziAPISyncSetValueI, 300, 570
ziAPISyncSetValueString, 302, 572
ziAPISyncSetValueStringUnicode, 303, 573
ziAPIUnSubscribe, 322, 579
ziAPIUpdateDevices, 544
ziAPIVectorWrite, 379, 623
ziAPIVectorWriteBlock, 377, 621
ziAPIVectorWriteGetStatus, 378, 622
ziAPIWriteDebugLog, 339, 593
ZIConnection, 256, 448
ZIModuleEventPtr, 340, 448
ZIModuleHandle, 448
ziDAQ
 Installation, 103
 Requirements, 103
zoomFFT Module, 99-101