LabOne Programming Manual

Zurich Instruments

LabOne Programming Manual

Zurich Instruments AG

Publication date Revision 38200

Revision History

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

Part I. LabOne Programming Concepts

This part of the manual provides an overview of LabOne programming and deals with generic concepts that apply to any of Zurich Instruments' APIs.

Refer to:

- Chapter 1 for an overview of programming with Zurich Instruments LabOne.
- Chapter 2 for an overview to working with `ziCore` Modules.

Chapter 1. Introduction

This chapter briefly describes the different possibilities to interface with a Zurich Instruments device, other than via the LabOne User Interface or ziControl (HF2 Series only). 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.

As an example, the user can either work on a computer directly connected to the instrument via USB or remotely from a different computer on the network, away from their experimental setup. Then, on either computer, the user can configure and retrieve data from their instrument via a number of different software interfaces, i.e. via the web-based LabOne User Interface and/or their own custom programs. In this way the user can decide which connectivity setup and combination of interfaces best suits their experimental setup and data processing needs.

Refer to:

- Section 1.1 for an overview of the LabOne Software Architecture.
- Section 1.2 for a Comparison of the LabOne APIs.
- Section 1.3 for help Initializing a Connection to a Data Server.
- Section 1.4 for help Configuring and Obtaining Data from an Instrument.
- Section 1.5 for Instrument-Specific Considerations.

Note

New users could benefit by first familiarizing themselves with the instrument using the LabOne User Interface or ziControl; please refer to the appropriate user manual for your instrument for more details.

Note

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 Software Architecture

Zurich Instruments devices uses 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.

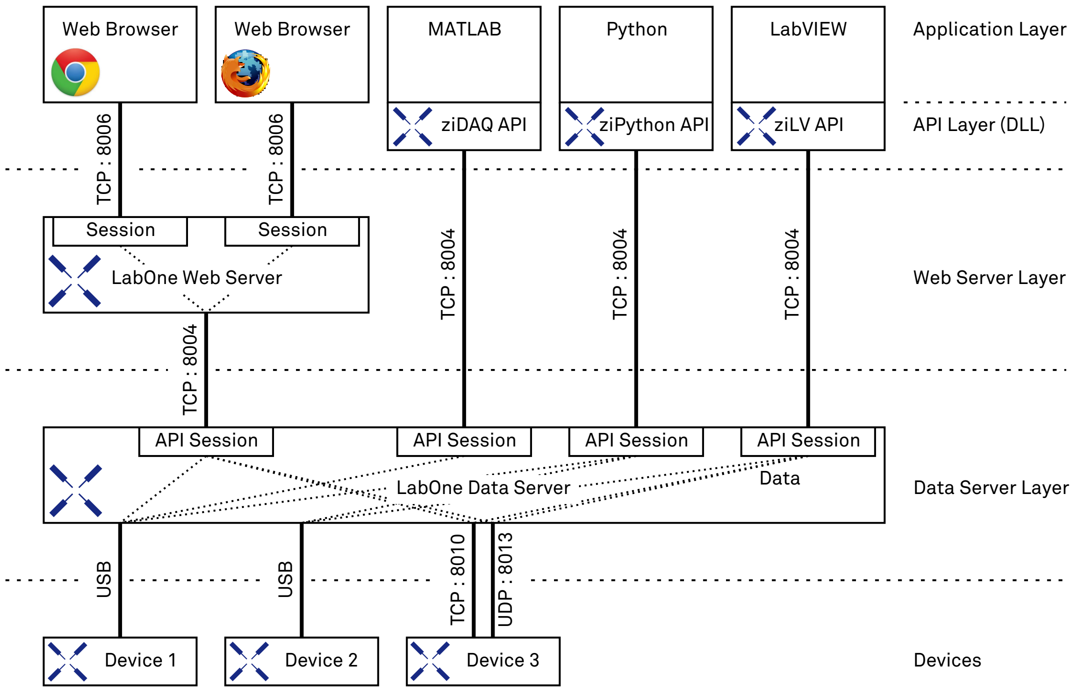

Figure 1.1. LabOne Software Architecture. The above diagram depicts the software architecture when using UHF and HF2 Instruments. In the case of MF Instruments the server runs on the device itself instead of on a PC; only one MF device can be accessed from the Data Server. Web Server and API usage for 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 UHFLI and MFLI 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 C++ library upon which many APIs are based, see Part II of this document.
- Modules: `ziCore` software components that provide a unified interface to APIs to perform a specific high-level common task such as sweeping data.

1.2. 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 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 high-level functionality provided by `ziCore`'s Sweep and Software Trigger Modules is also available in the LabVIEW and C APIs, all modules are available from the Matlab and Python LabOne APIs.

1.3. Initializing a Connection to a Data Server

As described in Section 1.1 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 in general the user's network topology). Users must be aware that HF2 instruments operate via a different data server program than other instruments (UHF, MF) and users of MF instruments should be aware that the data server runs on the MF instrument itself and not on a separate PC. Regardless of which data server is used and where it is running 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.

1.3.1. 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 UHF and MF 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 (UHF, HF2 instruments)

In the simplest configuration for 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 1.4.1 for more information on nodes.

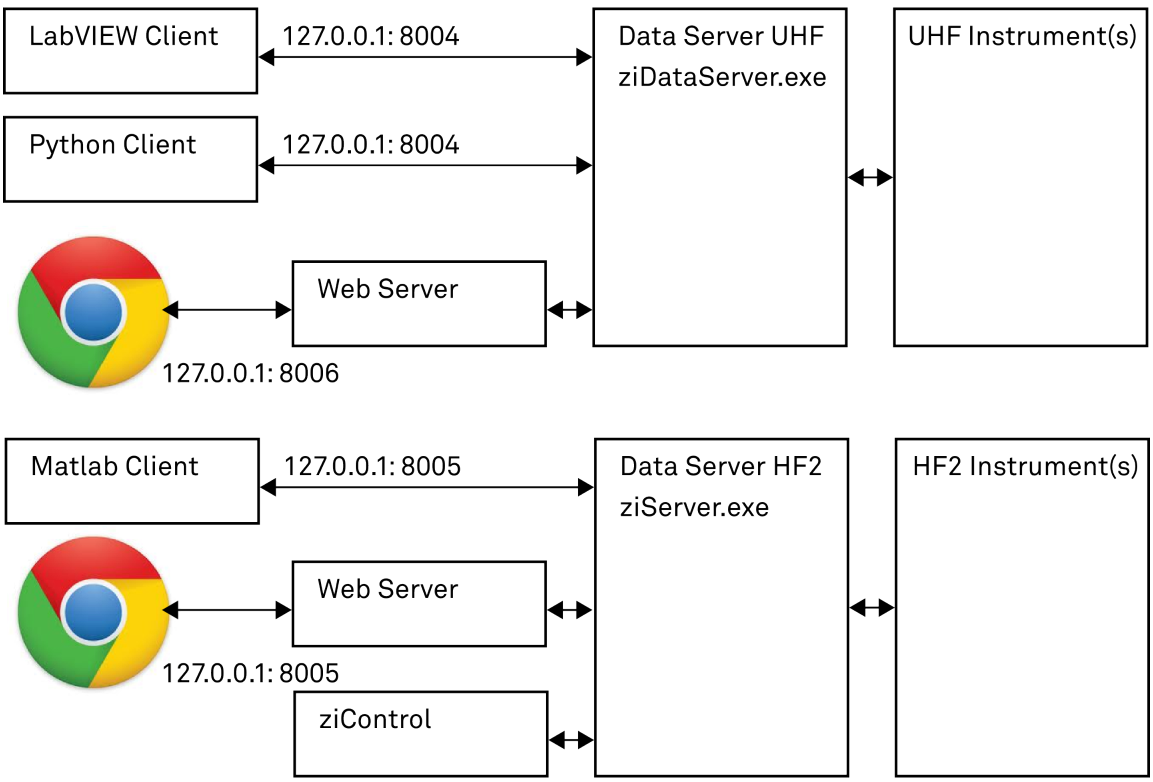

Figure 1.2. Server address and port handling for HF2 and UHF instruments for the case where the API client and data server are running on the same PC. In this case the server hostname is `localhost` and the default port value is 8004 for UHF Instruments and 8005 for HF2 Instruments.

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. Please see the Getting Started chapter of the MFLI User Manual for more details.

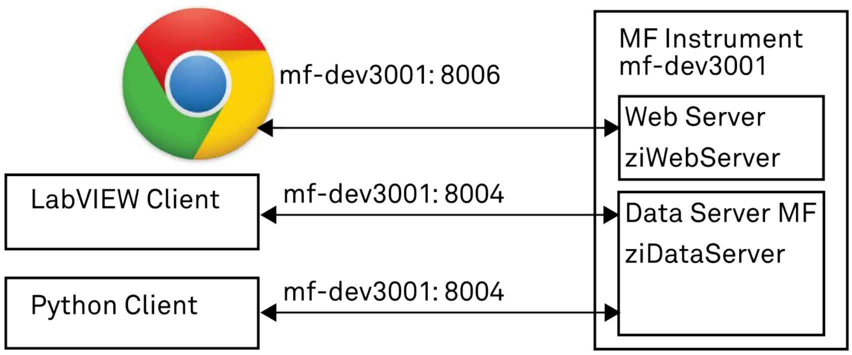

Figure 1.3. Server address and port handling on MF Instruments. Note, the data server is running on the instrument and the server hostname is the same as the instrument's hostname. The default data server port value is 8004 for MF Instruments. In this example, the MF has device serial dev3001.

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 of 5 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 1.3.2.

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 with device serial `'dev3001'` with the LabOne Matlab API:

```
>> ziDAQ('connect', 'mf-dev3001', 8004, 5)
```

1.3.2. 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.

Available API levels as of LabOne Software Release 15.01 are:

- API Level 1: HF2 support, basic UHF support.
- API Level 4: UHF support with timestamps and PWA, name clean-up.
- API Level 5: Introduction of scope offset for extended (non-hardware) scope inputs (UHF, MF Instruments).

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 UHFLI and MFLI API users are recommended to use API Level 5.

API Level 4 Features

The new features in API Level 4 are:

- Timestamps are available for any settings or data node.
- 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.

1.4. Configuring and Obtaining Data from an Instrument

1.4.1. Finding settings: The Node Hierarchy

In order to communicate with an Zurich Instruments device via text-based commands in an API, it is necessary to understand how the settings and measurement data of the instrument are accessed. All the settings and data of the instrument are organized in a file-system-like hierarchical structure. The features of the instrument, such as demodulators, are accessed as branches in this tree and their individual settings are leaves of these branches. It is also possible to browse branches inside the tree as if the user were navigating in a file-system. This hierarchy is used, no matter which interface you use when performing measurements.

An example demonstrating the hierarchy is the representation of the first demodulator on the device, given by the node:

```
/devX/demods/0
```

which, as we've already noted, is very similar to a **path** on a computer's file-system. Note that, the top level of the path is the device that you are connected to. The demodulators are then given as a top-level **node** under your device-node and the node of the first demodulator is indexed by `0`. This path represents a branch in the node hierarchy which, in this case, if we explore further, has the following nodes:

```
/devX/demods/0/adcselect
/devX/demods/0/order
/devX/demods/0/timeconstant
/devX/demods/0/rate
/devX/demods/0/trigger
/devX/demods/0/oscselect
/devX/demods/0/harmonic
/devX/demods/0/phaseshift
/devX/demods/0/sinc
/devX/demods/0/sample
```

These nodes are **leaves**, the most bottom-level nodes which represent a setting of an instrument or a field that can be read to retrieve measurement data. For example, `/devX/demods/0/adcselect` is the leaf that controls the setting corresponding to the choice of signal input for the first demodulator. To set the index of the signal input the user writes to this node. The leaf `/devX/demods/0/sample` is the leaf where the demodulator's output (timestamp, demodulated x-value, demodulated y-value) are written at the frequency specified by `/devX/demods/0/rate`. In order to obtain the demodulator output you read the values from this node by **polling** this node. Polling a node sends a request from the client to ziServer to obtain the data from the node at that particular point in time.

Note

The numbering on the front panel of the UHFLI, MFLI and HF2 Instruments and the block numbering in the graphical user interfaces generally start with 1 (1-based indexing). Note, that when accessing settings and data via a software interface, the numbering starting with 0 (0-based indexing).

Note

A useful method to learn about the nodes of your instrument is to look at the output of the history in the bottom of the graphical user interface. The status line always shows the last applied command and you can view the entire history by clicking the "Show Log" or "Show History" button. You will find paths like

```
/devx/sigins/0/ac = 1
```

after you switched on the AC mode for signal input 1, or

```
/devx/demods/1/rate = 7200.000000
```

after changing the readout rate of demodulator 2 to 7.2 kHz.

1.4.2. Obtaining Data from the Instrument

The subscribe and poll commands

The easiest way to obtain data from an instrument is via the `poll` command, available in all of the LabOne API interfaces. The `poll` command is a function for synchronous data recording from specified nodes of an instrument. Synchronous means that the interface is blocked during execution of the command, see Section 2.1.4 for asynchronous alternatives. `poll` takes two obligatory input arguments **recording time** and **timeout**.

The `subscribe` and `unsubscribe` commands are used to select the nodes from which data should be recorded. After subscribing to the node, the Data Server's internal data buffer will start filling with data from the subscribed nodes. The `poll` command will return the data that was recorded for the specified recording time (obligatory input argument) and any data that was already in the buffer since the last `poll`. To get rid of the data from earlier measurements it's possible to clear the buffer before polling by using the `flush` command.

In order to avoid losing data (the Data Server has a finite amount of memory available for its data buffers), long recording times (> 20s, depending on sampling rates and available memory) should be avoided. However, since internal data buffering on the Data Server ensures that no data is lost between `poll` commands, it's possible to record for longer periods of time by using the `poll` command inside a loop. In order to check that no data has been lost during a poll, the demodulator sample's `time` flags can be checked, see the section called “Demodulator Sample Data Structure”.

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 throw an error.

Note

One of the LabOne `ziCore` Modules could be a more efficient choice for data retrieval than the comparably low-level `poll` command, see the Section ziCore Modules in Part II.

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):

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

Note

Chapter 6 contains some details of other data structures.

1.5. Instrument-Specific Considerations

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

1.5.1. 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 `/devN/system/calib/calibrate` to 1 and then disabled using the `/devN/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 from ziCore-based APIs (i.e. Matlab or Python APIs) 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 2. ziCore Programming Overview

The LabOne APIs provide interfaces to configure, acquire data from, and run integral functionality of your Zurich Instruments device. These high-level interfaces are, however, just thin application layers based on a shared core API, `ziCore`. This chapter aims to describe the common functionality that's available to any of the interfaces (Matlab, Python, C, LabVIEW) based on `ziCore`.

Refer to:

- Section 2.1 for An Introduction to ziCore-based APIs.
- Section 2.2 for the Sweeper Module.
- Section 2.3 for the zoomFFT Module.
- Section 2.4 for the Software Trigger (Recorder) Module.
- Section 2.5 for the Device Settings Module.
- Section 2.6 for the PLL Advisor Module.
- Section 2.7 for some `ziCore` programming Tips and Tricks.

2.1. An Introduction to ziCore-based APIs

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 (Matlab, Python, LabVIEW and C).

2.1.1. 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 (Matlab, Python, LabVIEW 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 1.1 for an overview of the LabOne software architecture.

2.1.2. 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 2.1.

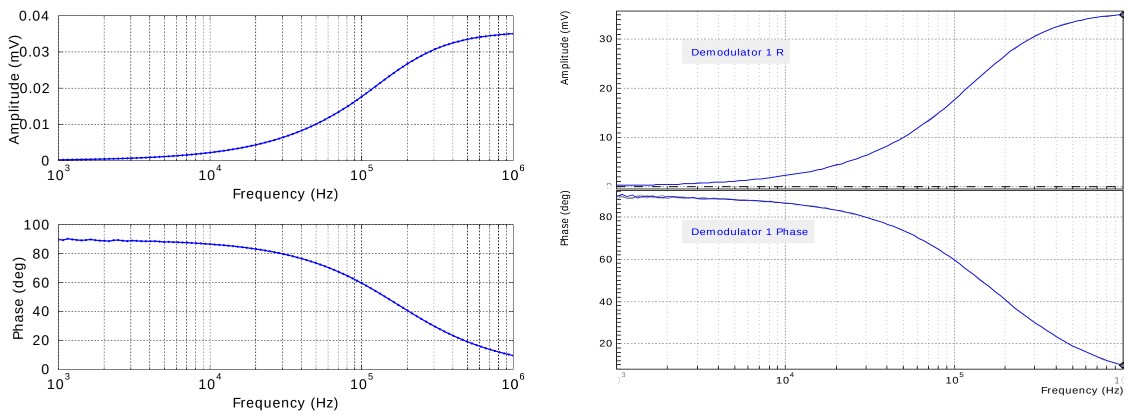

Figure 2.1. The same results and behavior can be obtained from Modules in any `ziCore`-based interface; Sweeper Module results from the LabOne Matlab API (left) and the LabOne User Interface (right) using the same Sweeper and instrument settings.

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 zoomFFT Module for calculating the FFT of demodulator output.

- The Software Trigger (Recorder) Module for recording instrument data asynchronously based upon user-defined triggers.
- The Device Settings Module for saving and loading instrument settings to and from (XML) files.
- The PLL Advisor Module for modeling/simulating the PLL (phase-locked loop) incorporated in the instrument (available for UHF Lock-in Amplifiers only).

In addition to providing a unified-interface between APIs, modules also provide a uniform work-flow regardless of the functionality the module performs (e.g., sweeping, recording data), see Section 2.1.3.

An important difference to low-level `ziCore` API commands is that Modules execute their commands **asynchronously**, see Section 2.1.4.

Note

The LabOne User Interface Command Log can be set to store commands in either Matlab or Python formats which can then be used to start writing custom programs, see Section 2.7.

Note

Much of the same functionality is provided in ziControl, but ziControl UI is not based on `ziCore`.

2.1.3. ziCore Module Work-Flow

Regardless of the Module's function, all `ziCore` Modules follow same work flow in all of the derived interfaces:

- create (instantiate) an instance of the module,
- **set** the module's parameters using `path`, `value` pairs,
- **subscribe** to instrument nodes from which to obtain data (note, this is a module subscribe, which is different from a normal API session subscribe command),
- **execute** the module (this starts the module's thread),
- wait until the module has **finished** executing; intermediate reading of data is possible,
- **read** the module's data,
- **clear** the module to remove it from memory.

The highlighted words above are commands for all the Modules. For interface-specific concepts when using Modules see the following Sections:

- Using ziCore Modules in the LabOne Matlab API,
- Using ziCore Modules in the LabOne Python API,
- Using ziCore Modules in the LabOne LabVIEW API,
- Using ziCore Modules in the LabOne C API.

2.1.4. 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.

2.2. 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.

2.2.1. Configuring the Sweeper

In the following we briefly describe how to configure the Sweeper Module. See Table 2.1 for a full list of the Sweeper's parameters and Table 2.2 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 2.4. 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 2.6. 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 2.5. 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 to 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 2.1 for recommended values of `sweep/settling/inaccuracy`. The relationship between `sweep/settling/inaccuracy` and `sweep/settling/tc` is plotted in Figure 2.2.

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 2.1. 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.

$$t_s = \max\left(\mathrm{sweep\_settling\_tc} \times \mathrm{tc},\ \mathrm{sweep\_settling\_time}\right)$$

Equation 2.1. 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).

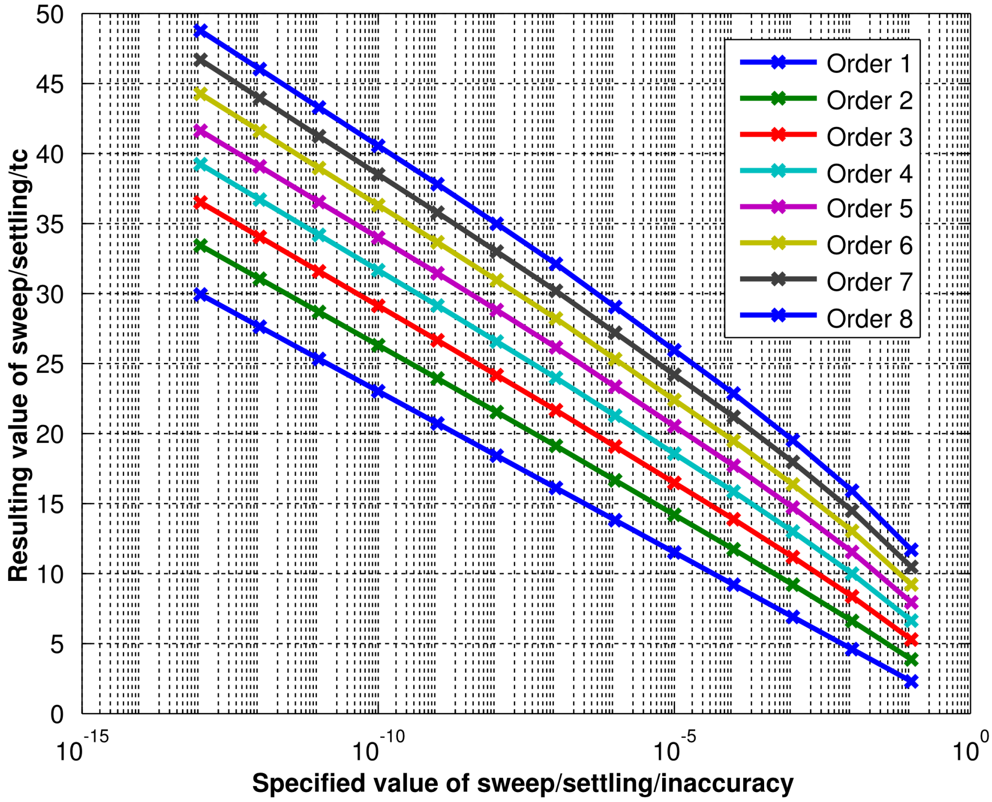

Figure 2.2. A plot showing the values of the Sweeper's `sweep/settling/tc` as calculated from `sweep/settling/inaccuracy` parameter and their dependency on demodulator order filter.

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 2.2.

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

Equation 2.2. 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 2.3 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 the `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 2.1. 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 → 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 2.2.

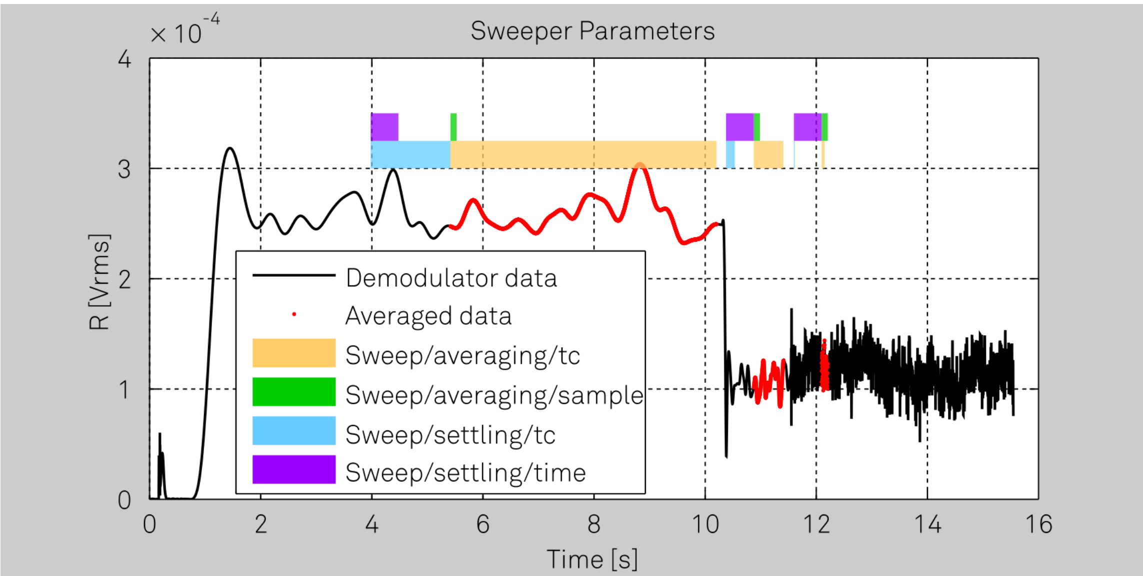

Figure 2.3. Plot demonstrating how the Sweeper records three measurement points from demodulator data when using automatic bandwidth control in a frequency sweep. Please see An Explanation of Settling and Averaging Times in a Frequency Sweep, above, for a detailed explanation.

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 (`rwr`) 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 2.1. Sweeper Parameters

Setting/Path	Type	Unit	Description
`sweep/device`	byte array	-	The device ID to perform the sweep on, e.g., `dev123` (compulsory parameter).
`sweep/gridnode`	byte array	Node	The device parameter (specified by node) to be swept, e.g., "oscs/0/freq".
`sweep/start`	double	Many	The start value of the sweep parameter.
`sweep/stop`	double	Many	The stop value of the sweep parameter.
`sweep/samplecount`	uint64	-	The number of measurement points to set the sweep on.
`sweep/endless`	bool	-	Enable Endless mode; run the sweeper continuously.
`sweep/remainingtime`	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.
`sweep/averaging/sample`	uint64	Samples	Sets the number of data samples per sweeper parameter point that is considered in the measurement. The maximum of this value and `sweep/averaging/tc` is taken as the effective calculation time. See Figure 2.3.
`sweep/averaging/tc`	double	Seconds	Sets the effective measurement time per sweeper parameter point that is considered in the measurement. The maximum between of this value and `sweep/averaging/sample` is taken as the effective calculation time. See Figure 2.3.
`sweep/bandwidthcontrol`	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 `sweep/bandwidth`); 2=Automatic. 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).

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 2.4); 1=Binary (Non-sequential sweep continues increase of resolution over entire range, see Figure 2.6), 2=Bidirectional (Sequential sweep from Start to Stop value and back to Start again, Figure 2.5), 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
			tc is taken as effective settling time. See Figure 2.3.
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/tc is plotted in Figure 2.2.
sweep/settling/tc	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 2.2.1 and Figure 2.2). The maximum between this value and sweep/settling/time is taken as effective settling time. See Figure 2.3.
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	byte array	-	The directory to which sweeper measurements are saved to via save().
sweep/savepath	byte array	-	This parameter is deprecated, see sweep/directory.
sweep/fileformat	byte array	-	The format of the file for saving sweeper measurements. 0=Matlab, 1=CSV.

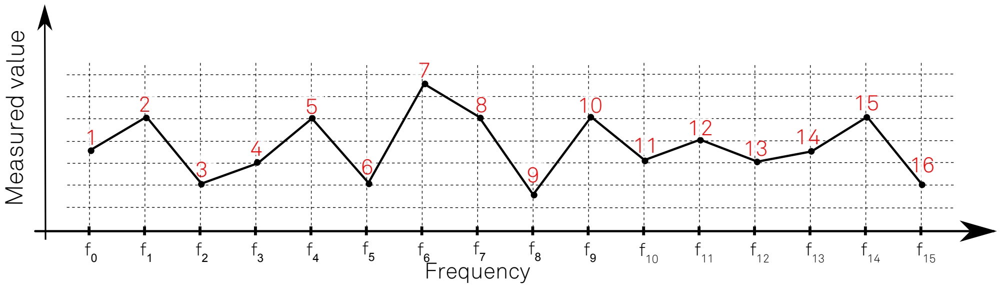

Figure 2.4. Sweeper scanning modes: Sequential (`sweep/scan` = 0).

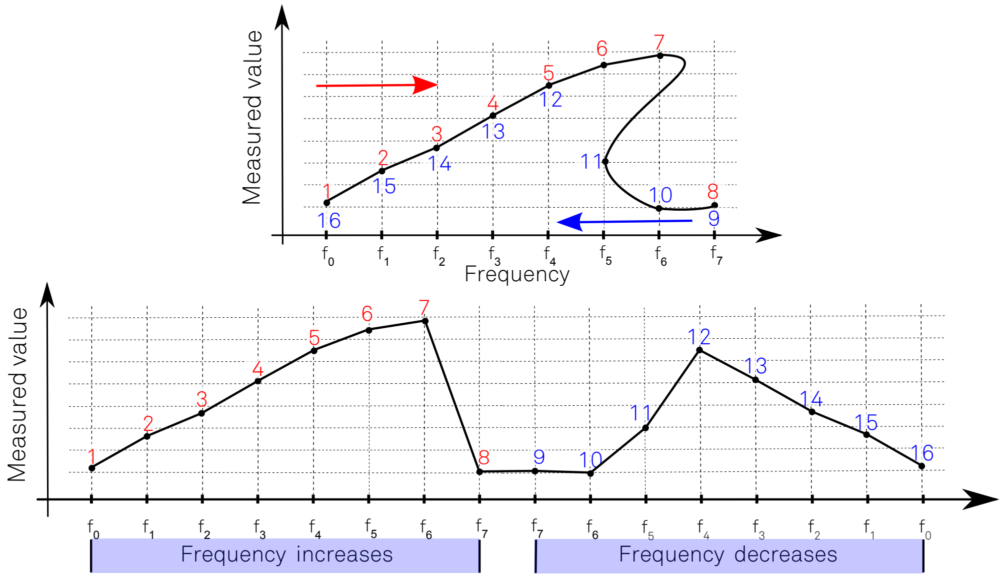

Figure 2.5. Sweeper scanning modes: Bidirectional (`sweep/scan` = 2).

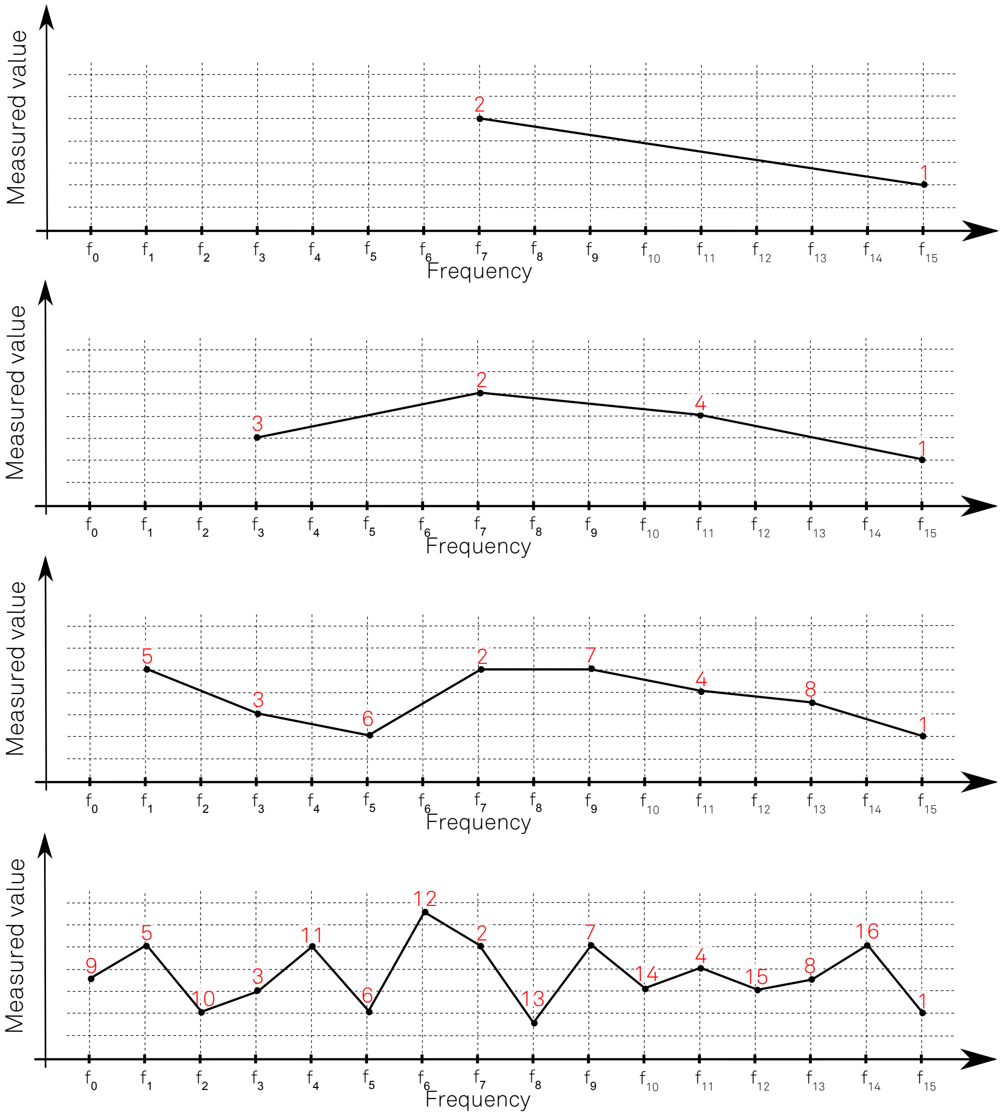

Figure 2.6. Sweeper scanning modes: Binary (`sweep/scan` = 1).

Table 2.2. 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.
`frequencystddev`	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
settimestamp	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., nexttimestamp - settimestamp corresponds roughly to the demodulator filter settling time.

2.3. zoomFFT Module

The zoomFFT Module corresponds to the Spectrum Tab of the LabOne User Interface. It allows the user to perform Fast Fourier Transforms (FFT) on a specified demodulator's output.

See Table 2.3 for the input parameters to configure the ZoomFFT Module and Table 2.4 for a description of the ZoomFFT's outputs.

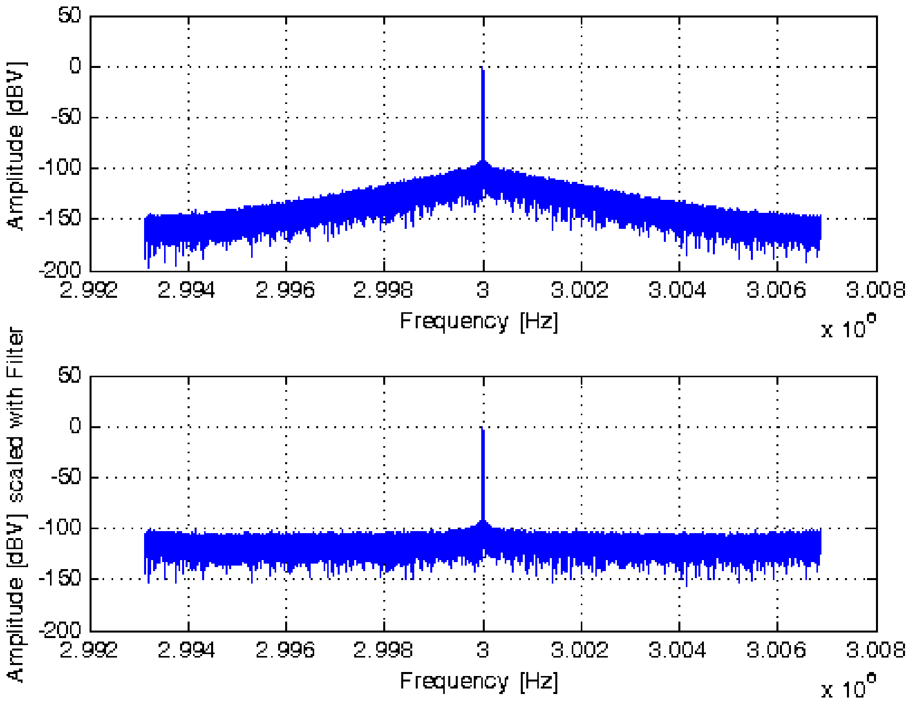

Figure 2.7. A plot of an FFT created by one of the LabOne Matlab API examples.

Table 2.3. ZoomFFT Input Parameters

Setting/Path	Type	Unit	Description
`zoomFFT/device`	byte array	-	The device ID to perform the FFT on, e.g., `dev123` (compulsory parameter).
`zoomFFT/absolute`	bool	-	Shifts the frequencies so that the center frequency becomes the demodulation frequency rather than 0 Hz.
`zoomFFT/bit`	uint64	-	Number of lines of the FFT spectrum (powers of 2). Increasing the bits increases the frequency resolution of the spectrum.
`zoomFFT/endless`	bool	-	Enable Endless mode; run the zoomFFT continuously.
`zoomFFT/loopcount`	uint64	-	The number of FFTs to perform.
`zoomFFT/mode`	uint64	-	Select the source signal for the FFT. 0=FFT(x +iy), 1=FFT(R), 2=FFT(phase), 3=FFT(Freq)
`zoomFFT/overlap`	double	-	Overlap of the demodulator data used for the FFT. Use 0 for no overlap and 0.99 for maximal overlap.

Setting/Path	Type	Unit	Description
`zoomFFT/settling/tc`	double	TC	Minimum wait time in factors of the time constant (TC) before starting the measurement. The maximum between this value and `zoomFFT/settling/time` is taken as effective settling time.
`zoomFFT/settling/time`	double	Seconds	Minimum wait time in seconds before starting the measurement. The maximum between 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 2.4. ZoomFFT Output Values

Name	Type	Unit	Description
`x`	double	Volts	The real part, x, of the complex FFT result.
`y`	double	Volts	The imaginary part, y, of the complex FFT result.
`r`	double	VoltsRMS	The absolute value, R, 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).
`rate`	double	-	Sampling rate of the demodulator.
`filter`	double	-	The filter envelope; the filter compensation value for each gridnode.
`bandwidth`	double	Hz	The bandwidth of the demodulator
`grid`	double	Hz	The frequency grid.
`nenbw`	double	-	The normalized equivalent noise bandwidth.
`resolution`	double	Hz	FFT resolution: Spectral resolution defined by the reciprocal acquisition time (sample rate, number of samples recorded).
`aliasingreject`	double	dB	How much damping is present at the border of your spectrum.

2.4. Software Trigger (Recorder) Module

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 2.8 for an example. The types of trigger available are listed in Table 2.5.

Table 2.5. 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 2.9.	1
Digital	Digital trigger with bit masking.	2
Pulse	Pulse width trigger with level hysteresis and noise reduction, see Figure 2.10 and Figure 2.11.	3
Tracking (edge or pulse)	Level tracking trigger to compensate signal drift, see Figure 2.12.	4
Hardware Trigger	UHFLI and MFLI only. Trigger on one of the instrument's hardware trigger channels.	6

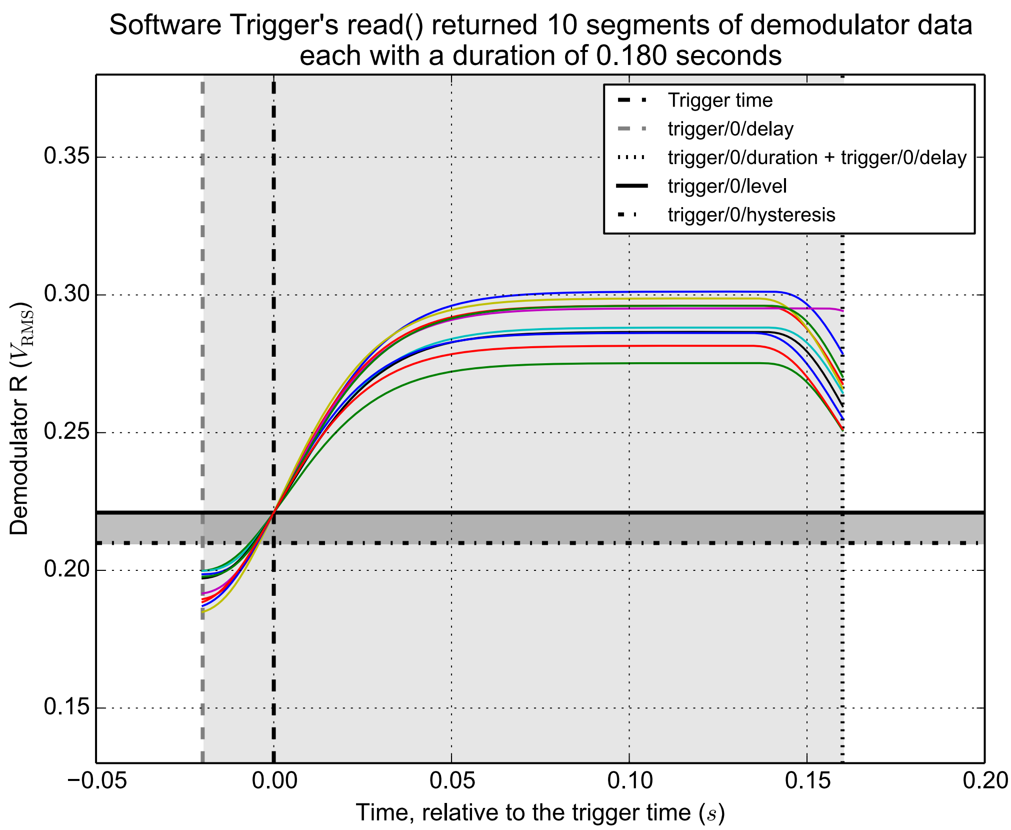

Figure 2.8. The plot produced by `example_swtrigger_edge.py`, an example distributed with the LabOne Python API. The plot shows 10 bursts of data from a single demodulator; each burst was recorded when the demodulator's R value exceeded a specified threshold using a positive edge trigger. See Section 4.2.3 for help getting started with the Python examples.

See Table 2.6 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.

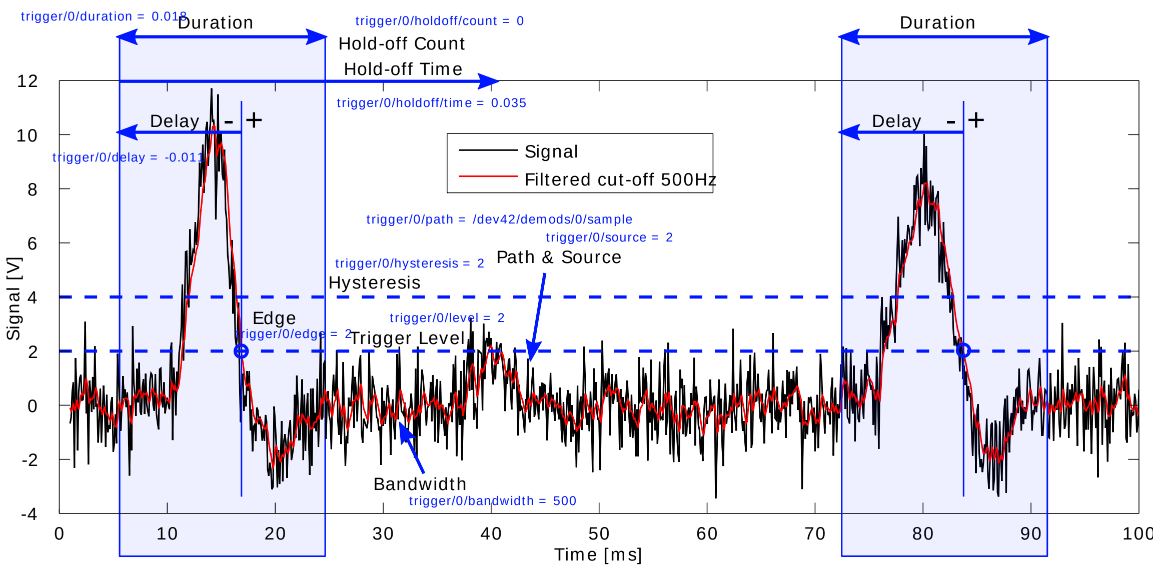

Figure 2.9. Explanation of the Software Trigger Module's parameters for an Edge Trigger.

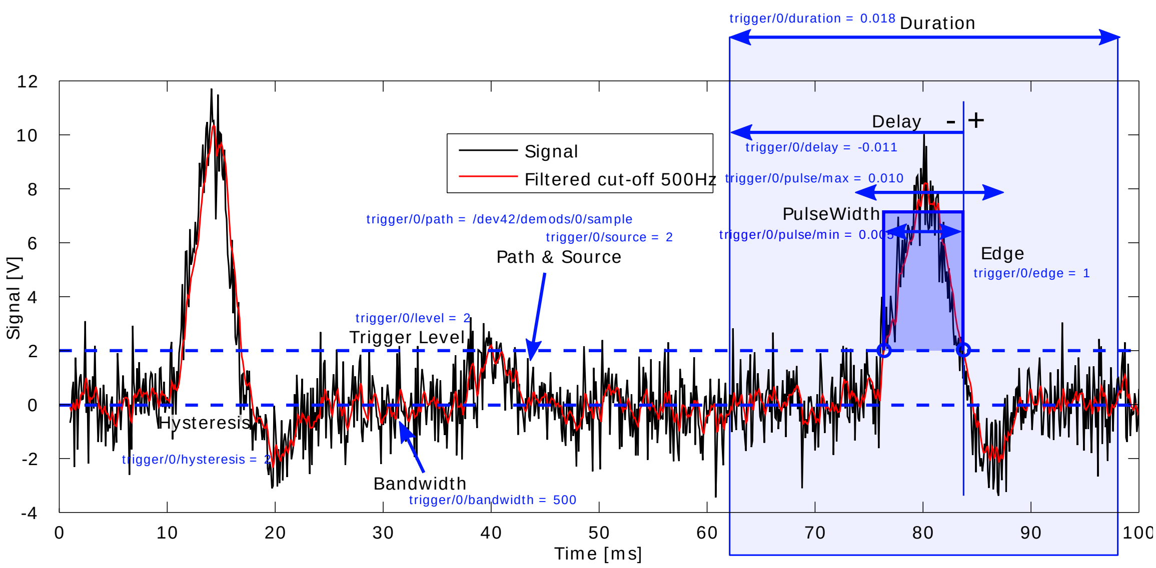

Figure 2.10. Explanation of the Software Trigger Module's parameters for a positive Pulse Trigger.

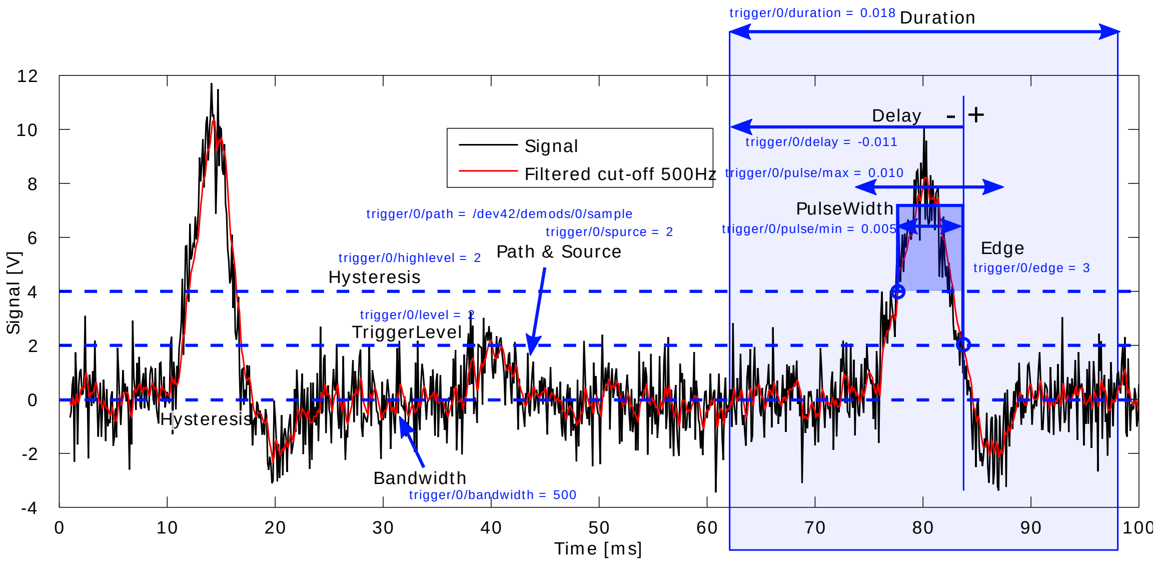

Figure 2.11. Explanation of the Software Trigger parameters for a positive or negative Pulse Trigger.

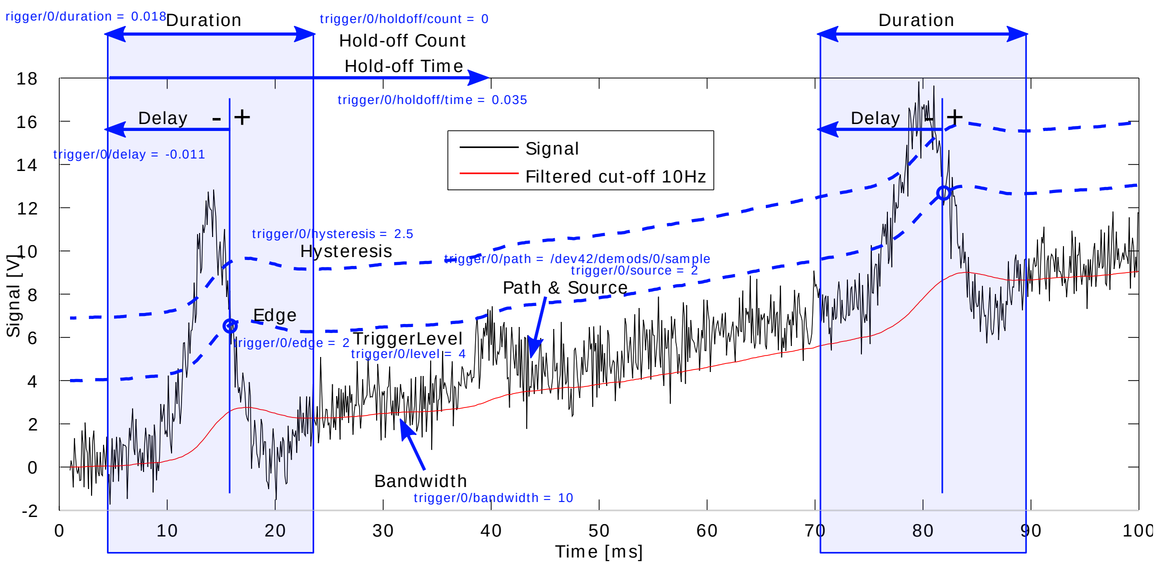

Figure 2.12. Explanation of the Software Trigger Module's parameters for a Tracking Trigger.

Table 2.6. Software Trigger Input Parameters.

Setting/Path	Type	Unit	Description
`trigger/device`	byte array	-	The device ID to execute the software trigger, e.g., `dev123` (compulsory parameter).
`trigger/buffersize`	double	Seconds	Set the buffer size of the trigger object. The recommended buffer size is 2*`trigger/N/duration`.
`trigger/flags`	uint64	-	Define the SW Trigger's behaviour if sampleloss is encountered: Fill holes (=0x01), align data that contains a timestamp (=0x02), throw EOFError if sampleloss is detected.

Setting/Path	Type	Unit	Description
`trigger/endless`	uint64	-	Enable endless triggering 1=enable; 0=disable.
`trigger/forcetrigger`	uint64	-	Force a trigger.
`trigger/filename`	byte array	-	This parameter is deprecated. If specified, i.e. not empty, it enables automatic saving of data in single triggering mode (`trigger/endless = 0`).
`trigger/savepath`	byte array	-	The directory where files are saved when saving data.
`trigger/fileformat`	byte array	-	The format of the file for saving data. 0=Matlab, 1=CSV.
`trigger/historylength`	uint64	-	Maximum number of entries stored in the measurement history.
`trigger/clearhistory`	uint64	-	Clear the measurement history
`trigger/triggered`	uint64	-	Has the software trigger triggered? 1=Yes, 0=No (read only).
`trigger/N/bandwidth`	double	Hz	Only for Tracking Triggers. The bandwidth used in the calculation of the exponential running average of the source signal.
`trigger/N/bitmask`	uint64	-	Only for Digital triggers. Specify the bitmask used with `trigger/N/bits`. The trigger value is bits AND bit mask (bitwise).
`trigger/N/bits`	uint64	-	Only for Digital triggers. Specify the bits used for the Digital trigger value. The trigger value is bits AND bit mask (bitwise)
`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 2.9.
`trigger/N/duration`	double	Seconds	The length of time to record data for, see Figure 2.9.
`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.

Setting/Path	Type	Unit	Description
`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 2.10.
`trigger/N/pulse/min`	double	-	Only for Pulse triggers: The minimum pulse width to trigger on. See Figure 2.10.
`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`	byte array	-	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 `SAMPLE.R` Demodulator Magnitude `SAMPLE.THETA` Demodulator Phase `SAMPLE.AUXIN0` Auxiliary Input 1 value `SAMPLE.AUXIN1` Auxiliary Input 2 value `SAMPLE.DIO` Digital I/O value Over HW Trigger paths may also be specified (device-class dependent). Overrides values from `trigger/0/path` and `trigger/0/source`.
`trigger/N/type`	uint64	-	The trigger type, see Table 2.5
`trigger/0/grid/mode`	int	-	Enable grid mode. In grid mode a matrix instead of a vector is returned by `read()`. 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.
`trigger/0/grid/operation`	int	-	If running in endless mode, either replace or average the data in the grid's matrix.

Setting/Path	Type	Unit	Description
`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/direction`	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.
`trigger/N/path`	byte array	-	This parameter is deprecated, see the `trigger/N/triggernode` parameter.
`trigger/N/source`	uint64	-	This parameter is deprecated, see the `trigger/N/triggernode` parameter.
`trigger/N/hwtrigsource`	uint64	-	This parameter is deprecated, see the `trigger/N/triggernode` parameter.

Note

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 2.10 and Figure 2.11.

2.5. 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 asynchronously, 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 a synchronously (non-blocking).

See Table 2.7 for the input parameters to configure the Device Settings Module.

Table 2.7. Device Settings Input Parameters

Setting/Path	Type	Description
`deviceSettings/device`	byte array	The device ID to save the settings for, e.g., `dev123` (compulsory parameter).
`deviceSettings/command`	byte array	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).
`deviceSettings/filename`	byte array	The name of the file to load or save to.
`deviceSettings/path`	byte array	The path containing the file to load from or save to.

Table 2.8. Device Settings Parameters for use only by the LabOne Web Server.

Setting/Path	Type	Description
`deviceSettings/throwonerror`	uint64	Throw an exception is there was error executing the command.
`deviceSettings/errortext`	byte array	The error text used in error messages.
`deviceSettings/finished`	uint64	The status of the command (read-only).

2.6. PLL Advisor Module

The PLL Advisor Module corresponds to the PLL Advisor section of the LabOne User Interface PLL tab. The PLL Advisor is a mathematical model of the PLL incorporated in the instrument and provides a convenient way to tune parameters to obtain an optimal feedback loop performance for the desired application.

Note

Note the PLL Advisor Module is only available for UHF Lock-in Amplifiers.

Table 2.9. PLL Advisor Parameters.

Setting/Path	Type	Unit	Description
`pllAdvisor/bode`	struct	-	Output parameter. Contains the resulting bode plot of the PLL simulation.
`pllAdvisor/calculate`	uint64	-	Issues a command for the PLL Advisor to calculate values. Set the value to 1 to start the calculation.
`pllAdvisor/center`	double	Hz	Center frequency of the PLL oscillator. The PLL frequency shift is relative to this center frequency.
`pllAdvisor/d`	double	Hz/deg s	The PID differential gain.
`pllAdvisor/demodbw`	double	Hz	The demodulator bandwidth to use for the PLL loop filter.
`pllAdvisor/i`	double	Hz/deg/s	The PID integral gain
`pllAdvisor/mode`	uint64	-	Sets the PLL operating mode. Currently only open-loop mode is supported.
`pllAdvisor/order`	uint64	-	Demodulator filter order to use for the PLL loop filter.
`pllAdvisor/p`	double	Hz/deg	The PID proportional gain.
`pllAdvisor/pllbw`	double	Hz	The demodulator bandwidth to use for the PLL loop filter.
`pllAdvisor/pm`	double	deg	Output parameter. Simulated phase margin of the PLL with the current settings. The phase margin should be greater than 45 deg and preferably greater than 65 deg for stable conditions.
`pllAdvisor/pmfreq`	double	-	Output parameter. Simulated phase margin frequency.
`pllAdvisor/q`	double	-	Quality factor. Currently not used.
`pllAdvisor/rate`	double	Hz	PLL Advisor sampling rate of the PLL control loop.
`pllAdvisor/stable`	bool	-	Output parameter. When 1, the PLL Advisor found a stable solution with the given settings. When 0, revise your settings and rerun the PLL Advisor.

Setting/Path	Type	Unit	Description
`pllAdvisor/targetbw`	double	Hz	Requested PLL bandwidth. Higher frequencies may need manual tuning.
`pllAdvisor/targetfail`	bool	-	Output parameter. 1 indicates the simulated PLL BW is smaller than the Target BW.

2.7. Tips and Tricks

Use the LabOne User Interface's Command Log to start programming

If you use the LabOne User Interface to perform a measurement, you can obtain the commands sent to your instrument in the "Command Log" by clicking the "Show Log" button in the status bar at the bottom of the User Interface. Be sure to set the "Log Format" of the Command Log in the "User Interface" section of the Config Tab first: The log is available in Matlab and Python formats and can be used as a starting point for your own custom program.

Use the included examples to get started programming

Both the LabOne Matlab API and the LabOne Python API come with examples to help you get started programming. In particular, both APIs have at least one example for each of the ziCore modules.

Load LabOne User Interface settings files from the APIs.

The XML files used for device settings can be loaded and saved from the LabOne User Interface or from any of the `ziCore`-based APIs. This means that an instrument can be conveniently configured via the LabOne User Interface and then its settings saved to file. This settings file can then be loaded via an API in order to configure an instrument for a script. See the Section 2.5.

Use the API's logging capabilities.

All of the LabOne APIs can write a log which can contain useful debugging or status information. See the relevant section in the API's chapter for more details:

- Enabling Logging in the LabOne Matlab API,
- Enabling Logging in the LabOne Python API,
- Error Handling and Logging in the LabOne C API.

Part II. LabOne APIs

This part of the Programming Manual documents language-specific installation and usage for each of the LabOne APIs. For details of common functionality and features that are shared by all the LabOne APIs please refer to Part I.

Refer to:

- Chapter 3 for the LabOne Matlab API (`ziDAQ`).
- Chapter 4 for the LabOne Python API (`ziPython`).
- Chapter 5 for the LabOne LabVIEW API.
- Chapter 6 for the LabOne C API (`ziAPI`).

Chapter 3. 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 3.1 for help Installing the LabOne Matlab API.
- Section 3.2 for help Getting Started with the LabOne Matlab API and Running the Examples.
- Section 3.3 for some LabOne Matlab API Tips and Tricks.
- Section 3.4 for help Troubleshooting the LabOne Matlab API.
- Section 3.5 for LabOne Matlab API (ziDAQ) Command Reference.

Note

This section and the provided examples are no substitute for 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.

3.1. Installing the LabOne Matlab API

3.1.1. Requirements

To use LabOne's Matlab API, `ziDAQ`, a Matlab installation and license on either Windows or Linux is required. On Windows, Matlab R2009b (or newer) is required, both 32-bit and 64-bit platforms are supported. On Linux, Matlab R2014b (or newer) and a 64-bit platform is required. No additional Matlab Toolboxes are required to use `ziDAQ`.

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.

3.1.2. Windows or Linux

No additional installation steps are required to use `ziDAQ` on either Windows or Linux; it's only necessary to add the folder containing LabOne's Matlab API library 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/
   ```

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

   ```
   >> ziAddPath;
   ```

   On Windows (similar for Linux) 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 1and 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 3.1.3.

3.1.3. 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 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 1.1 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 1.3.1). In the Matlab command window type one of the following:

      - `>> ziDAQ('connect', 'localhost', 8005) % 8005 for HF2 Series`
      - `>> ziDAQ('connect', 'localhost', 8004, 5) % 8004 for UHFLI`
      - `>> ziDAQ('connect', mf-hostname, 8004, 5) % 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 1.3.1 for information about port choice and 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 1.3.1 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.

3.2. Getting Started with the LabOne Matlab API

This section introduces the user to the LabOne Matlab API.

3.2.1. 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 3.1.2 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
%   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
%   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
%   ziTC2BW            - Convert demodulator timeconstants to 3 dB Bandwidth
%
% EXAMPLES/COMMON - Examples that will run on any Zurich Instruments Device
%   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_pid_advisor               - Setup and optimize a PID for internal
%                                       PLL mode
%   example_poll                      - Record demodulator data using
%                                       ziDAQServer's synchronous poll function
%   example_record_async              - Record data asyncronously using ziDAQ's
%                                       record module
```

```
%   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_scope                       - Record scope data using ziDAQServer's
%                                         synchronous poll function
%   example_spectrum                    - Perform an FFT using ziDAQ's zoomFFT
%                                         module (Spectrum Tab of the LabOne UI)
%   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_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.
%
% EXAMPLES/UHF - Examples specific to the UHF Series
%   uhf_example_boxcar              - Record boxcar data using ziDAQServer's
%                                     synchronous poll function
%   uhf_example_scope_offset        - Record scope/digitizer data using
%                                     ziDAQServer's synchronous poll function
%
% EXAMPLES/HF2 - Examples specific to the HF2 Series
%   hf2_example_autorange             - determine and set an appropriate range
%                                       for a sigin 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
```

Note

On Windows the MEX-file is called either `ziDAQ.mexw64` or `ziDAQ.mexw32` for 64-bit and 32-bit platforms respectively and on Linux it is called `ziDAQ.mexa64` or `ziDAQ.mexa32`. If more than one MEX-file is present, Matlab automatically selects the correct MEX-file for the current platform.

3.2.2. 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 3.5, LabOne Matlab API (ziDAQ) Command Reference for a printer friendly version.

3.2.3. Running the Examples

Prerequisites for running the Matlab examples:

1. Matlab is configured for `ziDAQ` as described above in Section 3.1.
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 3.2.1 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 3.1 for an example. If you encounter an error message please ensure that the above prerequisites are fulfilled and see Section 3.4 for help troubleshooting the error.

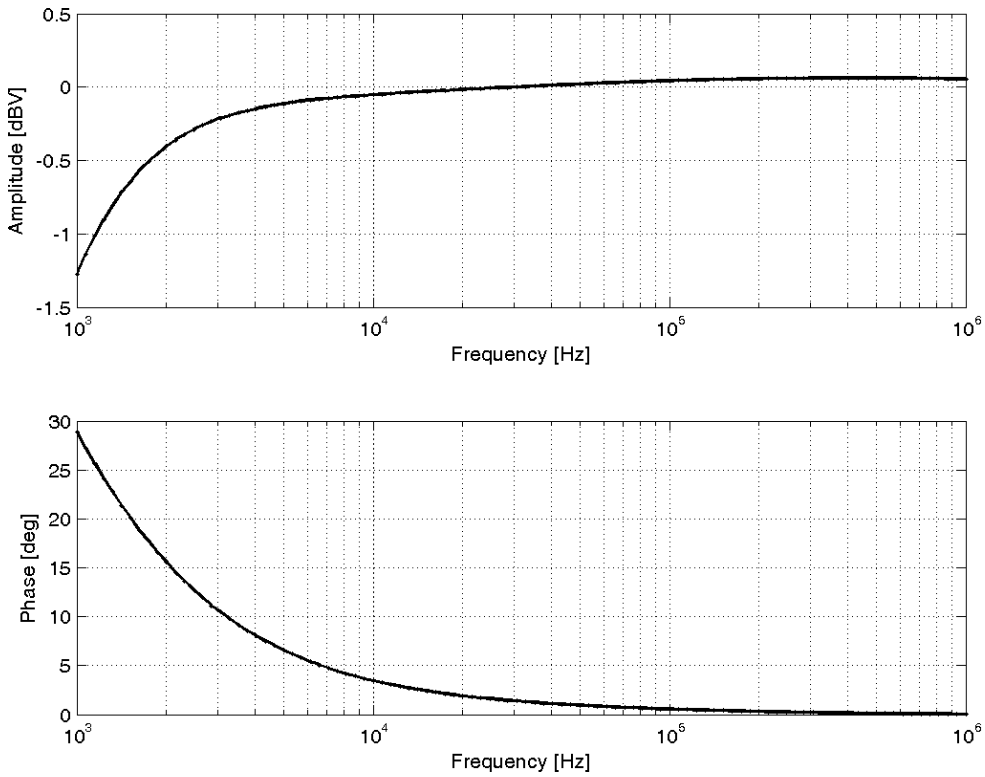

Figure 3.1. The plot produced by the LabOne Matlab API example `example_sweeper.m`; the plots show the instruments demodulator output when performing a frequency sweep over a simple feedback cable.

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.

3.2.4. 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 2.1.2 for more help with Modules and a description of their parameters.

3.2.5. 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.

3.3. 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 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...
```

3.4. 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 3.1.2.

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

Matlab can not find the example. Check whether the `MATLAB2012/Examples/Common` subfolder (respectively `MATLAB2012/Examples/UHF` or `MATLAB2012/Examples/HF2`) 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 3.1.2.

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 1.3.1.

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);
  ```

  where `hostname` should be replaced by the IP of the computer the Data Server is running on and `port` is specified as in Section 1.3.1.

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);
```

where `hostname` should be replaced by the IP address of the computer the Data Server is running on and `port` is specified as in Section 1.3.1. 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 3.1 and navigate away from the `Driver` directory. Secondly, ensure that the LabOne Matlab MEX-file is in the `Driver` folder as described in Section 3.2.1.

3.5. LabOne Matlab API (ziDAQ) Command Reference

```
%
% Copyright 2009-2016, 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 = 'clear', 'connect', 'connectDevice',
%                  'disconnectDevice', 'finished', 'flush', 'get',
%                  'getAsEvent', 'getAuxInSample','getByte',
%                  'getDIO', 'getDouble', 'getInt',
%                  'getSample', 'listNodes', 'logOn', 'logOff',
%                  'poll', 'pollEvent', 'programRT', 'progress', 'read',
%                  'record', 'setByte', 'setDouble', 'syncSetDouble',
%                  'setInt', 'syncSetInt', 'subscribe',
%                  'sweep', 'trigger', 'unsubscribe', 'update',
%                  'zoomFFT', 'deviceSettings'
%
% 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.
%
%            ziDAQ('connectDevice', device, interface);
%                  device (string) = Device serial to connect (e.g. 'DEV2000')
%                  interface (string) = Interface, e.g., 'USB', '1GbE', '10GbE'.
%                  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_LEAFSONLY 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
%                  Flags may also be combined, e.g., set flags to bitor(1, 2)
%                  to return paths recursively and printed as absolute paths.
%
%   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('getInt', path);
%                  path (string) = Node path
%
%   result = ziDAQ('getByte', path);
%                  path (string) = Node path
%
%            ziDAQ('setDouble', path, value);
%                  path (string) = Node path
%                  value (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('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, [settginsOnly]);
%                  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.
%                  [settginsOnly] (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 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) = Recording time in [s]
%                  timeout (int64) = Poll timeout in [ms]
%                  [flags] (uint32) = Flags that specify data polling properties
%                            Bit[0] FILL : Fill data loss holes
%                            Bit[1] ALIGN : Align data of several demodulators
%                            Bit[2] THROW : Throw if data loss is detected
%                  Records data for the specified time. This function call
%                  is blocking. Use ziDAQRecorder's asynchronous interface for
%                  long recording durations.
%
%   result = ziDAQ('pollEvent', timeout);
%                  timeout (int64) = Poll timeout in [ms]
%                  Execute a single poll command. This is a low-level
%                  function. The poll function is better suited for most
%                  cases.
%
% Asynchronous Interface
%
%   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.
%     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/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 = hardware trigger on trigger line source
%                                   6 = tracking edge trigger on source
%                                   7 = 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/hwtrigsource  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/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/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/filename        string This parameter is deprecated. If specified,
%                                    i.e., not empty, it enables automatic saving of
%                                    data in single trigger mode
%                                    (trigger/endless = 0).
%     trigger/directory       string The directory where files are saved.
%     trigger/fileformat      int    The format of the file for saving data.
%                                    0 = Matlab,
%                                    1 = CSV,
%                                    2 = ZView (Impedance data only).
%     trigger/historylength   bool   Maximum number of entries stored in the
%                                    measurement history.
%     trigger/clearhistory    bool   Remove all records from the history list.
%
%   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 class (note that
%                  the module's thread is not yet started) 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.
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  handle = Matlab handle (reference) specifying an instance of
%                           the ziDAQRecorder 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 ziDAQRecorder class.
%                    path (string) = Node path to record data from.
%                    Subscribe to device nodes. Call multiple times to
%                    subscribe to multiple node paths. After subscription the
%                    recording process can be started with the 'execute'
%                    command. During the recording process paths can not be
%                    subscribed or unsubscribed.
%
%              ziDAQ('unsubscribe', handle, path);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
%                    path (string) = Node path to record data from. Use wildcard
%                             ('*') to select all.
%                    Unsubscribe from one or several nodes. During the
%                    recording process paths can not be subscribed or
%                    unsubscribed.
%
%              ziDAQ('get', handle, path);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
%                    path (string) = Path string of the module parameter. Must
%                             start with 'sweep/'.
%                    Get module parameters. Wildcards are supported, e.g. 'sweep/*'.
%
%              ziDAQ('set', handle, path, value);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
%                    path (string) = Path string of the module parameter. Must
%                             start with 'sweep/'.
%                    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 ziDAQRecorder class.
%                    Start the recorder. After that command any trigger will
%                    start the measurement. Subscription or unsubscription
%                    is not possible until the recording is finished.
%
%              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.
%
%     result = ziDAQ('finished', handle);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
%                    Returns 1 if the recording is finished, otherwise 0.
%
%     result = ziDAQ('read', handle);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
%                    Read out the recorded data; transfer the recorded data to
%                    Matlab.
%
%              ziDAQ('finish', handle);
%                    handle = Matlab handle (reference) specifying an instance of
%                             the ziDAQRecorder class.
```

```
%                  Stop recording data. The recording may be restarted by
%                  calling 'execute' again.
%
%   result = ziDAQ('progress', handle);
%                  handle = Matlab handle (reference) specifying an instance of
%                           the ziDAQRecorder 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 ziDAQRecorder class.
%                  Stop the module's thread.
%
% Sweep Module
%
%   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/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/filename          string This parameter is deprecated. If specified,
%                                   i.e. not empty, it enables automatic saving of
%                                   data in single sweep mode (sweep/endless = 0).
%     sweep/directory         string The directory where files are located when
%                                   saving sweeper measurements.
%     sweep/fileformat        int    The format of the file for saving sweeper
%                                   measurements:
%                                   0 = Matlab,
%                                   1 = CSV,
%                                   2 = ZView (Impedance data only).
%     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)
%
%   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.
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  path (string) = Module parameter path
%                  flags (int64) = Define which module parameters paths should be
%                          returned, set the following bits to obtain the
%                          described behaviour:
%                            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);
%                  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.
%                  handle = Reference to the ziDAQSweeper class.
%                  path = Path string of the node. Use wild card to
%                  select all. Alternatively also a list of path
%                  strings can be specified.
%
%            ziDAQ('unsubscribe', handle, path);
%                  Unsubscribe from one or several nodes. During the
%                  recording process paths can not be subscribed or
%                  unsubscribed.
%                  handle = Reference to the ziDAQSweeper class.
%                  path = Path string of the node. Use wild card to
%                  select all. Alternatively also a list of path
%                  strings can be specified.
%
%            ziDAQ('execute', handle);
%                  Start the sweep. Subscription or unsubscription
%                  is no more possible until the sweep is finished.
%
%   result = ziDAQ('finished', handle);
%                  handle = Handle of the sweep session.
%                  Returns 1 if the sweep is finished, otherwise 0.
%
%   result = ziDAQ('read', handle);
%                  handle = Handle of the sweep session.
%                  Transfer the sweep data to Matlab.
%
%   result = ziDAQ('progress', handle);
%                  Report the progress of the measurement with a number
%                  between 0 and 1.
%
%            ziDAQ('finish', handle);
%                  Stop the sweep. The sweep may be restarted by
%                  calling 'execute' again.
%
%            ziDAQ('clear', handle);
%                  handle = Handle of the sweep session.
%                  Stop the current sweep.
```

```
%
%            ziDAQ('save', handle);
%                  Save the measured data to a file.
%                  handle = Handle of the sweep session.
%                  [filename] =  File in which to store the data.
%
% Zoom FFT Module
%
%   Zoom FFT Parameters
%     zoomFFT/device        string  Device that should be used for
%                                   the zoom 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
%     zoomFFT/loopcount     int     Number of zoom FFT loops (default 1)
%     zoomFFT/endless       int     Perform endless zoom FFT (default 0)
%                                   0 = Use loopcount value
%                                   1 = Endless zoom FFT enabled, ignore
%                                       loopcount
%     zoomFFT/overlap       double  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
%     zoomFFT/absolute      bool    Shifts the frequencies so that the center
%                                   frequency becomes the demodulation frequency
%                                   rather than 0 Hz.
%
%   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.
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  path (string) = Module parameter path
%                  flags (int64) = Define which module parameters paths should be
%                          returned, set the following bits to obtain the
%                          described behaviour:
%                            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);
```

```
%                   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.
%                   handle = Reference to the ziDAQZoomFFT class.
%                   path = Path string of the node. Use wild card to
%                   select all. Alternatively also a list of path
%                   strings can be specified.
%
%             ziDAQ('unsubscribe', handle, path);
%                   Unsubscribe from one or several nodes. During the
%                   recording process paths can not be subscribed or
%                   unsubscribed.
%                   handle = Reference to the ziDAQZoomFFT class.
%                   path = Path string of the node. Use wild card to
%                   select all. Alternatively also a list of path
%                   strings can be specified.
%
%             ziDAQ('execute', handle);
%                   Start the zoom FFT. Subscription or unsubscription
%                   is no more possible until the zoomFFT is finished.
%
%   result = ziDAQ('finished', handle);
%                   handle = Handle of the zoom FFT session.
%                   Returns 1 if the zoom FFT is finished, otherwise 0.
%
%   result = ziDAQ('read', handle);
%                   handle = Handle of the zoom FFT session.
%                   Transfer the zoomFFT data to Matlab.
%
%   result = ziDAQ('progress', handle);
%                   Report the progress of the measurement with a number
%                   between 0 and 1.
%
%             ziDAQ('finish', handle);
%                   Stop the zoomFFT. The zoom FFT may be restarted by
%                   calling 'execute' again.
%
%             ziDAQ('clear', handle);
%                   handle = Handle of the zoom FFT session.
%                   Stop the current zoom FFT.
%
% Device Settings Module
%
%   Device Settings Parameters
%     deviceSettings/device         string  Device whose settings are to be
%                                           saved/loaded, e.g. 'dev99'.
%     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).
%
%   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 the thread start, set the path,
%                   filename and command parameters. To run the command, use the
%                   execute function.
```

```
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  path (string) = Module parameter path
%                  flags (int64) = Define which module parameters paths should be
%                          returned, set the following bits to obtain the
%                          described behaviour:
%                            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);
%                  Not relevant for the device settings module.
%
%            ziDAQ('unsubscribe', handle, path);
%                  Not relevant for the device settings module.
%
%            ziDAQ('execute', handle);
%                  Execute the command.
%
%   result = ziDAQ('finished', handle);
%                  handle = Handle of the device settings session.
%                  Returns 1 if the command is finished, otherwise 0.
%
%   result = ziDAQ('read', handle);
%                  handle = Handle of the device settings session.
%                  Transfer the device settings to Matlab.
%                  Not relevant since device settings are saved to a file.
%
%   result = ziDAQ('progress', handle);
%                  Report the progress of the command with a number
%                  between 0 and 1.
%
%            ziDAQ('finish', handle);
%                  Stop the device settings module. The module may be restarted by
%                  calling 'execute' again.
%
%            ziDAQ('clear', handle);
%                  handle = Handle of the device settings session.
%                  End the current device settings thread.
%
% PLL Advisor Module
%
%   PLL Advisor Parameters
%     pllAdvisor/bode       struct  Output parameter. Contains the resulting bode
%                                   plot of the PLL simulation.
%     pllAdvisor/calculate  int     Command to calculate values. Set to 1 to start
%                                   the calculation.
%     pllAdvisor/center     double  Center frequency of the PLL oscillator. The PLL
%                                   frequency shift is relative to this center
%                                   frequency.
%     pllAdvisor/d          int     Differential gain.
%     pllAdvisor/demodbw    int     Demodulator bandwidth used for the PLL loop
%                                   filter.
%     pllAdvisor/i          double  Integral gain.
```

```
%     pllAdvisor/mode       double  Select PLL Advisor mode. Currently only one mode
%                                   (open loop) is supported.
%     pllAdvisor/order      double  Demodulator order used for the PLL loop filter.
%     pllAdvisor/p          int     Proportional gain.
%     pllAdvisor/pllbw      int     Demodulator bandwidth used for the PLL loop
%                                   filter.
%     pllAdvisor/pm         int     Output parameter. Simulated phase margin of the
%                                   PLL with the current settings. The phase margin
%                                   should be greater than 45 deg and preferably
%                                   greater than 65 deg for stable conditions.
%     pllAdvisor/pmfreq     int     Output parameter. Simulated phase margin
%                                   frequency.
%     pllAdvisor/q          int     Quality factor. Currently not used.
%     pllAdvisor/rate       int     PLL Advisor sampling rate of the PLL control
%                                   loop.
%     pllAdvisor/stable     int     Output parameter. When 1, the PLL Advisor found
%                                   a stable solution with the given settings. When
%                                   0, revise your settings and rerun the PLL
%                                   Advisor.
%     pllAdvisor/targetbw   int     Requested PLL bandwidth. Higher frequencies may
%                                   need manual tuning.
%     pllAdvisor/targetfail int     Output parameter. 1 indicates the simulated PLL
%                                   BW is smaller than the Target BW.
%
%   handle = ziDAQ('pllAdvisor', timeout);
%                  timeout = Poll timeout in [ms] - DEPRECATED, ignored
%                  Creates a PLL Advisor class for simulating the PLL in the
%                  device. Before the thread start, set the command parameters,
%                  call execute() and then set the "calculate" parameter to start
%                  the simulation.
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  path (string) = Module parameter path
%                  flags (int64) = Define which module parameters paths should be
%                          returned, set the following bits to obtain the
%                          described behaviour:
%                            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);
%                  Subscribe to one or several nodes.
%
%            ziDAQ('unsubscribe', handle, path);
%                  Unsubscribe from one or several nodes..
%
%            ziDAQ('execute', handle);
%                  Start the PLL Advisor.
%
%   result = ziDAQ('finished', handle);
%                  handle = Handle of the PLL Advisor session.
%                  Returns 1 if the command is finished, otherwise 0.
%
%   result = ziDAQ('read', handle);
```

```
%                  handle = Handle of the PLL Advisor session.
%                  Read pllAdvisor data. If the simulation is still ongoing only a
%                  subset of the data is returned.
%
%   result = ziDAQ('progress', handle);
%                  Report the progress of the command with a number
%                  between 0 and 1.
%
%            ziDAQ('finish', handle);
%                  Stop the PLL Advisor module.
%
%            ziDAQ('clear', handle);
%                  handle = Handle of the PLL Advisor session.
%                  End the current PLL Advisor thread.
%
% 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/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.
%
%   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.
%
%   result = ziDAQ('listNodes', handle, path, flags);
%                  handle = Handle of the PID Advisor session.
%                  path (string) = Module parameter path
%                  flags (int64) = Define which module parameters paths should be
%                          returned, set the following bits to obtain the
%                          described behaviour:
%                            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 settings
%                  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 = Handle of the PID Advisor session.
%                  Subscribe to one or several nodes.
%
```

```
%            ziDAQ('unsubscribe', handle, path);
%                  handle = Handle of the PID Advisor session.
%                  Unsubscribe from one or several nodes..
%
%            ziDAQ('get', handle, path);
%                  handle = Handle of the PID Advisor session.
%                  path = Path string of the node.
%
%            ziDAQ('execute', handle);
%                  handle = Handle of the PID Advisor session.
%                  Starts the pidAdvisor if not yet running.
%
%            ziDAQ('trigger', handle);
%                  Not applicable to this module.
%
%   result = ziDAQ('finished', handle);
%                  handle = Handle of the PID Advisor session.
%                  Returns 1 if the command is finished, otherwise 0.
%
%   result = ziDAQ('read', handle);
%                  handle = Handle of the PID Advisor session.
%                  Read pidAdvisor data. If the simulation is still ongoing only a
%                  subset of the data is returned.
%
%   result = ziDAQ('progress', handle);
%                  handle = Handle of the PID Advisor session.
%                  Report the progress of the command with a number
%                  between 0 and 1.
%
%            ziDAQ('finish', handle);
%                  handle = Handle of the PID Advisor session.
%                  Stop the PID Advisor module.
%
%            ziDAQ('clear', handle);
%                  handle = Handle of the PID Advisor session.
%                  End the current PID Advisor thread.
%
%            ziDAQ('save', handle);
%                  Save the measured data to a file.
%                  handle = Handle of the PID Advisor session.
%                  [filename] =  File name string (without extension)..
%
% 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_SYNC_SET_DOUBLE:  0x00000010
%                          LOG_SYNC_SET_INT:     0x00000020
%                          LOG_SYNC_SET_BYTE:    0x00000040
%                          LOG_GET_DOUBLE:       0x00000100
%                          LOG_GET_INT:          0x00000200
%                          LOG_GET_BYTE:         0x00000400
%                          LOG_GET_DEMOD:        0x00001000
```

```
%                          LOG_GET_DIO:          0x00002000
%                          LOG_GET_AUXIN:        0x00004000
%                          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.
%
```

Chapter 4. Python Programming

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 expensive software licenses. The Zurich Instruments LabOne Python API, also known as `ziPython` enables the user to configure and stream data from their instrument directly into Python.

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

- Section 4.1 for help Installing the LabOne Python API.
- Section 4.2 for help Getting Started with the LabOne Python API and Running the Examples.
- Section 4.3 for LabOne Python API Tips and Tricks.
- Section 4.4 for the LabOne Python API (ziPython) Command Reference.

Note

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.

4.1. Installing the LabOne Python API

4.1.1. Requirements

In order to install and use the LabOne Python API you require:

1. Either a Python 2.7 or a Python 3.5 installation on either Windows or Linux.
2. The NumPy python package installed for your Python installation.
3. The correct version of `ziPython` for your Python version and platform, available from the Zurich Instruments download page.

Note

Linux users must also ensure they download the version of `ziPython` that is Unicode compatible with their Linux distribution's Python installation, see Section 4.1.4 for help determining which version is required.

Note

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”.

4.1.2. 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 `ziPython`'s examples.
2. SciPy - recommended to load data saved from the LabOne UI in binary Matlab format (.mat).

Note

Unofficial pre-compiled 32-bit and 64-bit Windows binaries of NumPy, SciPy and matplotlib are available from Christoph Gohlke's pythonlibs page.

4.1.3. Windows Installation

To install `ziPython` on Windows execute the `.msi` installer available from the Zurich Instruments download page. It will guide you through the installation process as displayed in the following screenshots.

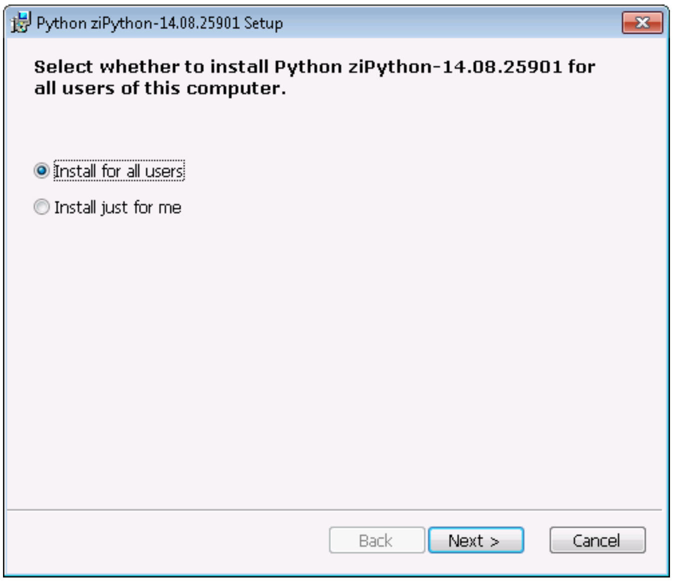

Figure 4.1. Windows `ziPython` installation: Step 1.

If multiple Python Installations are available on your system, the installer will ask which Python version the `ziPython` package should be installed. The `ziPython` package will be installed in selected versions in the folder `[PYTHONROOT]\lib\site-packages\zhinst\`.

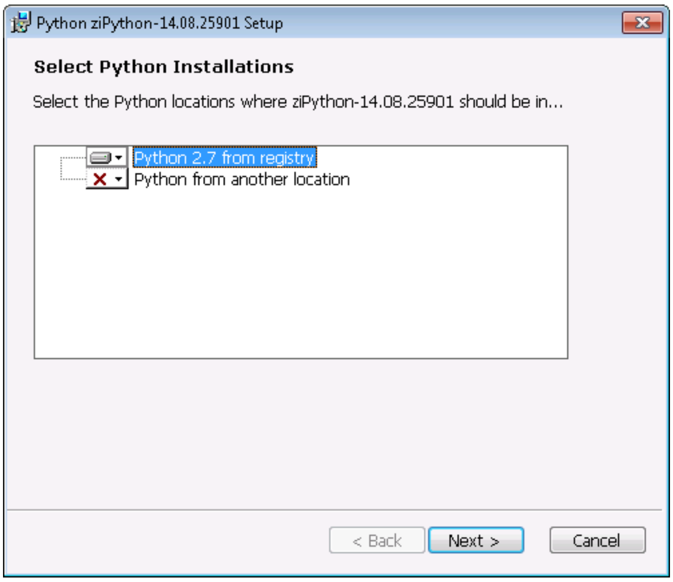

Figure 4.2. Windows `ziPython` installation: Step 2.

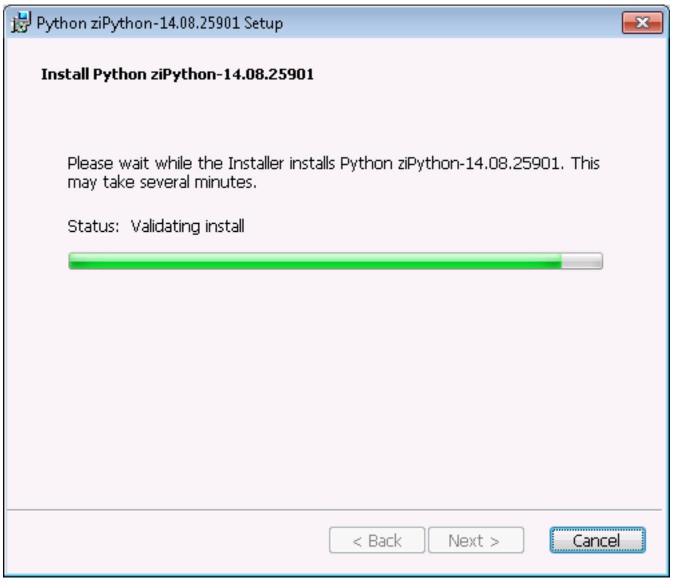

Figure 4.3. Windows `ziPython` installation: Step 3.

4.1.4. Linux Installation

In addition to the requirements above and selecting the correct version of `ziPython` for your Python distribution (2.7 or 3.5) and platform (32-bit or 64-bit), on Linux the correct Unicode version must also be installed. This is because some Python distributions on Linux are compiled to use UCS-2 character encoding, whereas some use UCS-4.

Determining the correct Unicode version of **`ziPython`** for your Python distribution

In order to determine which version of Unicode your Python 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 UCS-2 version of `ziPython`,
- `1114111`, use the UCS-4 version of `ziPython`.

Note

The installation needs root access rights. If you do not have these permissions, ask your system administrator for help.

To install `ziPython` on a Debian-derived distribution such as Ubuntu perform the following steps:

1. If required, install Python, NumPy and matplotlib (with elevated access rights):

   ```
   $ sudo apt-get install python python-numpy python-matplotlib
   ```

2. Unpack the `ziPython` software bundle:

   ```
   $ tar xzf ziPython-[version]-[build]-[linux32|linux64].tar.gz
   ```

3. Change directory into the unpacked folder and run the setup script `setup.py` as following:

```
$ cd ziPython-[version]-[build]-(linux32|linux64)
$ python setup.py build
$ sudo python setup.py install --install-layout=deb # Elevated access rights
```

   It's possible to skip the `build` step (`install` will automatically perform this step), but splitting the steps avoids creating a directory in your user space which is owned by `root`.

4.2. Getting Started with the LabOne Python API

This section introduces the user to the LabOne Python API.

4.2.1. Contents of the LabOne Python API

Alongside the driver for interfacing with your Zurich Instruments device, the LabOne Python API includes utility functions and examples. See:

- Section 4.4.1 to see which examples are available in `ziPython`.
- Section 4.4.2 to see which utility functions are available in `ziPython`.

4.2.2. 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.ziDAQSweeper)
```

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

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

See Section 4.4, LabOne Python API (ziPython) Command Reference for a printer friendly version of the built-in documentation.

4.2.3. Running the Examples

Prerequisites for running the Python examples:

1. The `zhinst` package is installed 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 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 4.1.2.

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.uhf`: examples only compatible with the UHF Lock-in Amplifier,
- `zhinst.examples.hf2`: examples only compatible with HF2 Series Instruments.

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 `ziPython` example is to import the example's module in an interactive shell and call the `run_example()` function. For example, to run the zoomFFT Module example:

```
>>> import zhinst.examples
>>> # Use do_plot=False if matplotlib is unavailable
>>> zhinst.examples.common.example_spectrum.run_example('dev123', do_plot=True);
```

The example should produce some output in the Python shell, such as:

```
Will perform 1 zoomFFTs
Individual zoomFFT 100.00 complete.
sample contains 1 zoomFFTs
Number of lines in first zoomFFT: 65535
```

Most examples will also plot the retrieved data using matplotlib, see Figure 4.4 for an example. If you encounter an error message please ensure that the above prerequisites are fulfilled.

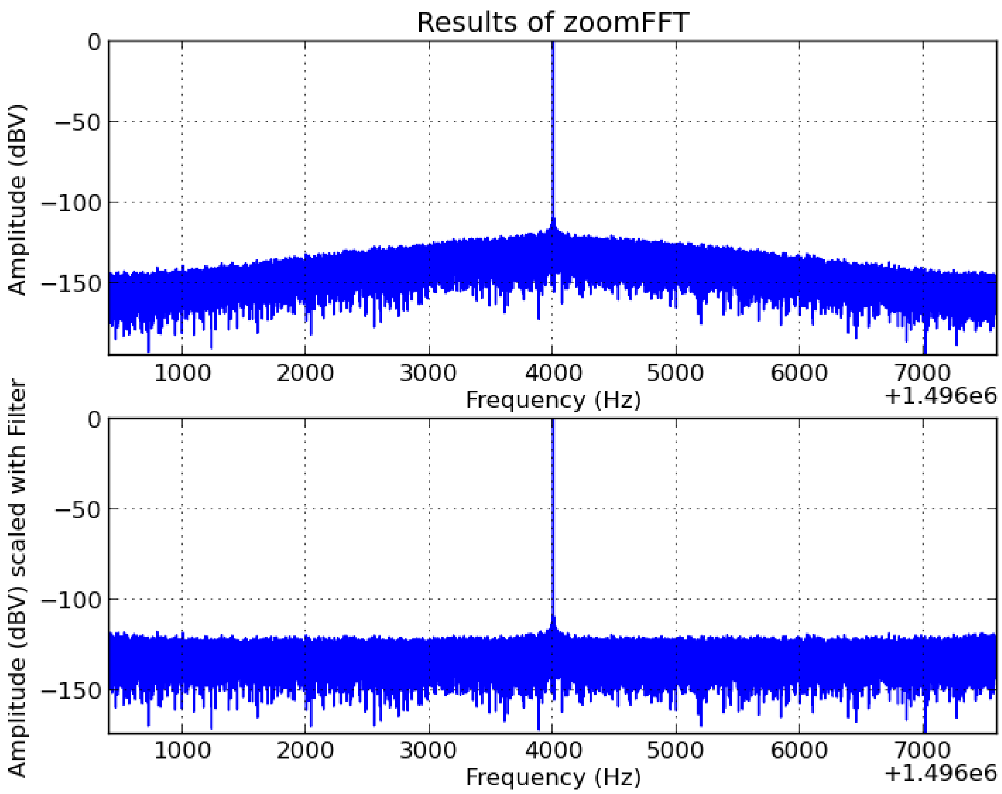

Figure 4.4. The plot produced by the LabOne Python API example `example_spectrum.py`; the plots show the results of an FFT performed with `ziCore`'s zoomFFT module on demodulator output obtained over a simple feedback cable.

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. For example, for 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_connect
    example_connect_config
    example_pid_advisor
    example_poll
    example_record_edge_trigger
    example_save_device_settings_expert
    example_save_device_settings_simple
    example_scope
    example_spectrum
    example_sweeper
    example_swtrigger_edge
    example_swtrigger_grid
    example_swtrigger_trackingedge
    example_zoomfft

DATA
    __all__ = ['example_connect', 'example_connect_config', 'example_pid_a...

FILE
    /home/ci/.pyenv/versions/3.5.1/lib/python3.5/site-packages/zhinst/examples/
common/__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,

```
>>> import zhinst
>>> print zhinst.__path__
```

will output something similar to:

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

4.2.4. Using ziCore Modules in the LabOne Python API

In the LabOne Python API `ziCore` Modules are configured and controlled by instantiating an object of the Module's class. For example, in order to use the Sweeper Module a sweeper object is created as following:

```
>>> daq = ziPython.ziDAQServer('localhost', 8004, 5) # Create a connection to the
 Data Server
      (running on the same PC as the API client)
>>> 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 `ziDAQSweeper` 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 2.1.2 for more help with Modules and a description of their parameters.

4.2.5. 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`'s log 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.

4.3. 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 4.4.2 for reference documentation on these commands.

4.4. 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 4.2.2) command; It is included here for convenience.

The documentation is grouped by module and class as following:

4.4.1. 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/3.5.1/lib/python3.5/site-packages/zhinst/__init__.py
```

4.4.2. 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
    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
    converstion for the provided demodulator order.

check_for_sampleloss(timestamps)
    Check whether timestamps are equidistantly spaced, it 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 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)

   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.plt 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.plt 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)
```

```
sigin_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.sigin_autorange(daq, device_serial, input_channel)

    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')
    logger = <logging.Logger object>
    print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0)...

FILE
    /home/ci/.pyenv/versions/3.5.1/lib/python3.5/site-packages/zhinst/utils.py
```

4.4.3. 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
 |
 |  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>(...)
|
|   awgModule(...)
|       awgModule( (ziDAQServer)arg1) -> ziAwgModule :
|           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
|
|   deviceSettings(...)
|       deviceSettings( (ziDAQServer)arg1) -> ziDeviceSettings :
|           Create a deviceSettings class. This will start a thread for running an
|           asynchronous deviceSettings.
|               arg1: Reference to the ziDAQServer class.
|               arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED,
ignored
|
|       deviceSettings( (ziDAQServer)arg1, (int)arg2) -> ziDeviceSettings
|
|   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.
 |
 |  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
 |
 |  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.
 |
 |  getList(...)
 |      getList( (ziDAQServer)arg1, (str)arg2) -> 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.
 |
 |  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
 |
 |  impedanceModule(...)
 |      impedanceModule( (ziDAQServer)arg1) -> ziImpedanceModule :
 |          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 ziDAQRecorder 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
 |                    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_SYNC_SET_DOUBLE:  0x00000010
 |                           LOG_SYNC_SET_INT:     0x00000020
 |                           LOG_SYNC_SET_BYTE:    0x00000040
 |                           LOG_GET_DOUBLE:       0x00000100
 |                           LOG_GET_INT:          0x00000200
 |                           LOG_GET_BYTE:         0x00000400
 |                           LOG_GET_DEMOD:        0x00001000
 |                           LOG_GET_DIO:          0x00002000
 |                           LOG_GET_AUXIN:        0x00004000
 |                           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
|
|   pidAdvisor(...)
|       pidAdvisor( (ziDAQServer)arg1) -> ziPidAdvisor :
|           Create a pidAdvisor class. This will start a thread for running an
|           asynchronous pidAdvisor.
|               arg1: Reference to the ziDAQServer class.
|               arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED,
ignored
|
|       pidAdvisor( (ziDAQServer)arg1, (int)arg2) -> ziPidAdvisor
|
|   pllAdvisor(...)
|       pllAdvisor( (ziDAQServer)arg1) -> ziPllAdvisor :
|           Create a pllAdvisor class. This will start a thread for running an
|           asynchronous pllAdvisor.
|               arg1: Reference to the ziDAQServer class.
|               arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED,
ignored
|
|       pllAdvisor( (ziDAQServer)arg1, (int)arg2) -> ziPllAdvisor
|
|   poll(...)
|       poll( (ziDAQServer)arg1, (float)arg2, (int)arg3, (int)arg4, (bool)arg5) ->
object :
|           This function returns subscribed data previously in the API's buffers or
|           obtained during the specified time. It returns a dict tree containing
|           the recorded data. This function blocks until the recording time is
|           elapsed.    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.
|                               ALIGN = 0x0002 : Align data that contains a
|                                                timestamp.
|                               THROW = 0x0004 : Throw EOFError exception if sample
|                                                loss is detected.
|               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 :
|           Execute a single poll command. Note: only one data packet will be
|           fetched. To get all data waiting in the buffers this command should be
|           executed continuously until nothing is returned anymore. This is a low
|           level command. Use the poll command or asynchronous recording instead.
|               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) -> ziDAQRecorder :
|          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.
|
|
|      record( (ziDAQServer)arg1, (float)arg2, (int)arg3 [, (int)arg4]) ->
ziDAQRecorder
|
|  revision(...)
|      revision( (ziDAQServer)arg1) -> int :
|          Get the revision number of the Python interface of Zurich Instruments.
|              arg1: Reference to the ziDAQServer class.
|
|  saveEngine(...)
|      saveEngine( (ziDAQServer)arg1) -> ziSaveEngine :
|          Create a saveEngine class. This will start a thread for running an
|          asynchronous saveEngine.
|              arg1: Reference to the ziDAQServer class.
|
|      saveEngine( (ziDAQServer)arg1, (int)arg2) -> ziSaveEngine
|
|  set(...)
|      set( (ziDAQServer)arg1, (object)arg2) -> None :
|              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.
|
|  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.
|
|  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) -> ziDAQSweeper :
|          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) -> ziDAQSweeper
|
|  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.
|
|  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) -> ziDAQZoomFFT :
|          Create a zoomFFT class. This will start a thread for running an
|          asynchronous zoomFFT.
|              arg1: Reference to the ziDAQServer class.
|              arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED,
ignored
|
|      zoomFFT( (ziDAQServer)arg1, (int)arg2) -> ziDAQZoomFFT
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __instance_size__ = 56
|
|  ----------------------------------------------------------------------
|  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__
```

4.4.4. Help for ziPython's **ziDeviceSettings** class

An instance of `ziDeviceSettings` 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) -> ziDeviceSettings :
        Create a deviceSettings class. This will start a thread for running an
        asynchronous deviceSettings.
            arg1: Reference to the ziDAQServer class.
            arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED, ignored

    deviceSettings( (ziDAQServer)arg1, (int)arg2) -> ziDeviceSettings
```

Reference help for the `ziDeviceSettings` class.

```
>>> help('zhinst.ziPython.ziDeviceSettings')

Help on class ziDeviceSettings in zhinst.ziPython:

zhinst.ziPython.ziDeviceSettings = class ziDeviceSettings(Boost.Python.instance)
 |  Method resolution order:
 |      ziDeviceSettings
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
```

```
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziDeviceSettings)arg1) -> None :
 |          End the deviceSettings thread.
 |
 |  execute(...)
 |      execute( (ziDeviceSettings)arg1) -> None :
 |          Execute the save/load command.
 |
 |  finish(...)
 |      finish( (ziDeviceSettings)arg1) -> None :
 |          Stop the load/save command. The command may be restarted by calling
 |          'execute' again.
 |
 |  finished(...)
 |      finished( (ziDeviceSettings)arg1) -> bool :
 |          Check if the command execution has finished. Returns True if finished.
 |
 |  get(...)
 |      get( (ziDeviceSettings)arg1, (str)arg2, (bool)arg3) -> object :
 |          Return a dict with all nodes from the specified sub-tree.
 |              arg1: Reference to the ziDeviceSettings 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( (ziDeviceSettings)arg1, (str)arg2) -> object
 |
 |  listNodes(...)
 |      listNodes( (ziDeviceSettings)arg1, (str)arg2, (int)arg3) -> list :
 |          This function returns a list of node names found at the specified path.
 |              arg1: Reference to the ziDeviceSettings 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( (ziDeviceSettings)arg1) -> object :
 |          Reports the progress of the command with a number between
 |          0 and 1.
 |
 |  read(...)
 |      read( (ziDeviceSettings)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( (ziDeviceSettings)arg1) -> object
 |
 |  save(...)
 |      save( (ziDeviceSettings)arg1, (str)arg2) -> None :
 |          Not relevant for the deviceSettings module.
 |
 |  set(...)
 |      set( (ziDeviceSettings)arg1, (str)arg2, (float)arg3) -> None :
 |          Device Settings Parameters
 |          Path name                Type    Description
 |          deviceSettings/device    string  Device that should be used for
 |                                           loading/saving device settings,
 |                                           e.g. 'dev99'.
 |          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( (ziDeviceSettings)arg1, (str)arg2, (int)arg3) -> None
 |
 |      set( (ziDeviceSettings)arg1, (str)arg2, (str)arg3) -> None
 |
 |      set( (ziDeviceSettings)arg1, (object)arg2) -> None :
 |              arg1: Reference to the ziDeviceSettings class.
 |              arg2: A list of path/value pairs.
 |
 |  subscribe(...)
 |      subscribe( (ziDeviceSettings)arg1, (str)arg2) -> None :
 |          Not relevant for the deviceSettings module.
 |
 |  trigger(...)
 |      trigger( (ziDeviceSettings)arg1) -> None :
 |          Not applicable to this module.
 |
 |  unsubscribe(...)
 |      unsubscribe( (ziDeviceSettings)arg1, (str)arg2) -> None :
 |          Not relevant for the deviceSettings module.
 |
 |  ----------------------------------------------------------------------
 |  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__
```

4.4.5. Help for ziPython's **ziDAQSweeper** class

An instance of `ziDAQSweeper` 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) -> ziDAQSweeper :
        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) -> ziDAQSweeper
```

Reference help for the `ziDAQSweeper` class.

```
>>> help('zhinst.ziPython.ziDAQSweeper')

Help on class ziDAQSweeper in zhinst.ziPython:

zhinst.ziPython.ziDAQSweeper = class ziDAQSweeper(Boost.Python.instance)
 |  Method resolution order:
 |      ziDAQSweeper
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziDAQSweeper)arg1) -> None :
 |          End the sweeper thread.
 |
 |  execute(...)
 |      execute( (ziDAQSweeper)arg1) -> None :
 |          Start the sweeper. Subscription or unsubscription is no more
 |          possible until the sweep is finished.
 |
 |  finish(...)
 |      finish( (ziDAQSweeper)arg1) -> None :
 |          Stop sweeping. The sweeping may be restarted by calling
 |          'execute' again.
 |
 |  finished(...)
 |      finished( (ziDAQSweeper)arg1) -> bool :
 |          Check if the sweep has finished. Returns True if finished.
 |
 |  get(...)
 |      get( (ziDAQSweeper)arg1, (str)arg2, (bool)arg3) -> object :
 |          Return a dict with all nodes from the specified sub-tree.
 |              arg1: Reference to the ziDAQSweeper 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( (ziDAQSweeper)arg1, (str)arg2) -> object
 |
 |  listNodes(...)
 |      listNodes( (ziDAQSweeper)arg1, (str)arg2, (int)arg3) -> list :
 |          This function returns a list of node names found at the specified path.
 |              arg1: Reference to the ziDAQRecorder 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( (ziDAQSweeper)arg1) -> object :
|          Reports the progress of the measurement with a number between
|          0 and 1.
|
|  read(...)
|      read( (ziDAQSweeper)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( (ziDAQSweeper)arg1) -> object
|
|  save(...)
|      save( (ziDAQSweeper)arg1, (str)arg2) -> None :
|          Save sweeper data to file.
|              arg1: Reference to the ziDAQSweeper class.
|              arg2: File name string (without extension).
|
|  set(...)
|      set( (ziDAQSweeper)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/endless          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
|                                        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/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/filename         string  This parameter is deprecated. If
|                                          specified, i.e. not empty, it enables
|                                          automatic saving of data in single sweep
|                                          mode (sweep/endless = 0).
|          sweep/directory        string  The directory where files are located
|                                          when saving sweeper measurements.
|          sweep/fileformat       string  The format of the file for saving sweeper
|                                          measurements:
|                                          0 = Matlab,
|                                          1 = CSV,
|                                          2 = ZView (Impedance data only).
|          sweep/historylength    bool    Maximum number of entries stored in the
|                                          measurement history.
|          sweep/clearhistory     bool    Remove all records from the history list.
|
|      set( (ziDAQSweeper)arg1, (str)arg2, (int)arg3) -> None
|
|      set( (ziDAQSweeper)arg1, (str)arg2, (str)arg3) -> None
|
|      set( (ziDAQSweeper)arg1, (object)arg2) -> None :
|              arg1: Reference to the ziDAQSweeper class.
|              arg2: A list of path/value pairs.
|
|  subscribe(...)
|      subscribe( (ziDAQSweeper)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 ziDAQSweeper 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( (ziDAQSweeper)arg1) -> None :
|          Execute a manual trigger.
|
|  unsubscribe(...)
|      unsubscribe( (ziDAQSweeper)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 ziDAQSweeper class.
|              arg2: Path string of the node. Use wild card to
|                    select all. Alternatively also a list of path
|                    strings can be specified.
|
|  ----------------------------------------------------------------------
|  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__
```

4.4.6. Help for ziPython's **ziDAQZoomFFT** class

An instance of `ziDAQZoomFFT` 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) -> ziDAQZoomFFT :
        Create a zoomFFT class. This will start a thread for running an
        asynchronous zoomFFT.
            arg1: Reference to the ziDAQServer class.
            arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED, ignored

    zoomFFT( (ziDAQServer)arg1, (int)arg2) -> ziDAQZoomFFT
```

Reference help for the `ziDAQZoomFFT` class.

```
>>> help('zhinst.ziPython.ziDAQZoomFFT')

Help on class ziDAQZoomFFT in zhinst.ziPython:

zhinst.ziPython.ziDAQZoomFFT = class ziDAQZoomFFT(Boost.Python.instance)
 |  Method resolution order:
 |      ziDAQZoomFFT
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziDAQZoomFFT)arg1) -> None :
 |          End the zoom FFT thread.
 |
 |  execute(...)
 |      execute( (ziDAQZoomFFT)arg1) -> None :
 |          Start the zoom FFT. Subscription or unsubscription is no more
 |          possible until the zoom FFT is finished.
 |
 |  finish(...)
 |      finish( (ziDAQZoomFFT)arg1) -> None :
 |          Stop the zoom FFT. The zoom FFT may be restarted by calling
 |          'execute' again.
 |
 |  finished(...)
 |      finished( (ziDAQZoomFFT)arg1) -> bool :
 |          Check if the zoom FFT has finished. Returns True if finished.
 |
```

```
|  get(...)
|      get( (ziDAQZoomFFT)arg1, (str)arg2, (bool)arg3) -> object :
|          Return a dict with all nodes from the specified sub-tree.
|              arg1: Reference to the ziDAQZoomFFT 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( (ziDAQZoomFFT)arg1, (str)arg2) -> object
|
|  listNodes(...)
|      listNodes( (ziDAQZoomFFT)arg1, (str)arg2, (int)arg3) -> list :
|          This function returns a list of node names found at the specified path.
|              arg1: Reference to the ziDAQZoomFFT 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( (ziDAQZoomFFT)arg1) -> object :
|          Reports the progress of the measurement with a number between
|          0 and 1.
|
|  read(...)
|      read( (ziDAQZoomFFT)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( (ziDAQZoomFFT)arg1) -> object
|
|  save(...)
|      save( (ziDAQZoomFFT)arg1, (str)arg2) -> None :
|          Save zoom FFT data to file.
|              arg1: Reference to the ziDAQZoomFFT class.
|              arg2: File name string (without extension).
|
|  set(...)
|      set( (ziDAQZoomFFT)arg1, (str)arg2, (float)arg3) -> None :
|          Zoom FFT Parameters
|          Path name             Type    Description
|          zoomFFT/device        string  Device that should be used for
|                                        the zoom 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
 |          zoomFFT/loopcount     int     Number of zoomFFT loops (default 1)
 |          zoomFFT/endless       int     Perform endless zoomFFT (default 0)
 |                                        0 = endless off, use loopcount
 |                                        1 = endless on, ignore loopcount
 |          zoomFFT/overlap       double  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
 |          zoomFFT/absolute      bool    Shifts the frequencies so that the center
 |                                        frequency becomes the demodulation
 |                                        frequency rather than 0 Hz.
 |
 |      set( (ziDAQZoomFFT)arg1, (str)arg2, (int)arg3) -> None
 |
 |      set( (ziDAQZoomFFT)arg1, (str)arg2, (str)arg3) -> None
 |
 |      set( (ziDAQZoomFFT)arg1, (object)arg2) -> None :
 |              arg1: Reference to the ziDAQZoomFFT class.
 |              arg2: A list of path/value pairs.
 |
 |  subscribe(...)
 |      subscribe( (ziDAQZoomFFT)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 ziDAQZoomFFT 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( (ziDAQZoomFFT)arg1) -> None :
 |          Execute a manual trigger.
 |
 |  unsubscribe(...)
 |      unsubscribe( (ziDAQZoomFFT)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 ziDAQZoomFFT class.
 |              arg2: Path string of the node. Use wild card to
 |                    select all. Alternatively also a list of path
 |                    strings can be specified.
 |
 |  ----------------------------------------------------------------------
 |  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__
```

4.4.7. Help for ziPython's **ziDAQRecorder** class

An instance of `ziDAQRecorder` 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) -> ziDAQRecorder :
        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.


    record( (ziDAQServer)arg1, (float)arg2, (int)arg3 [, (int)arg4]) -> ziDAQRecorder
```

Reference help for the `ziDAQRecorder` class.

```
>>> help('zhinst.ziPython.ziDAQRecorder')

Help on class ziDAQRecorder in zhinst.ziPython:

zhinst.ziPython.ziDAQRecorder = class ziDAQRecorder(Boost.Python.instance)
 |  Method resolution order:
 |      ziDAQRecorder
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziDAQRecorder)arg1) -> None :
 |          End the recording thread.
 |
 |  execute(...)
 |      execute( (ziDAQRecorder)arg1) -> None :
 |          Start the recorder. After that command any trigger will start
 |          the measurement. Subscription or unsubscription is no more
 |          possible until the recording is finished.
 |
 |  finish(...)
 |      finish( (ziDAQRecorder)arg1) -> None :
 |          Stop recording. The recording may be restarted by calling
 |          'execute' again.
 |
 |  finished(...)
 |      finished( (ziDAQRecorder)arg1) -> bool :
 |          Check if the recording has finished. Returns True if finished.
 |
```

```
|  get(...)
|      get( (ziDAQRecorder)arg1, (str)arg2, (bool)arg3) -> object :
|          Return a dict with all nodes from the specified sub-tree.
|              arg1: Reference to the ziDAQRecorder 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( (ziDAQRecorder)arg1, (str)arg2) -> object
|
|  listNodes(...)
|      listNodes( (ziDAQRecorder)arg1, (str)arg2, (int)arg3) -> list :
|          This function returns a list of node names found at the specified path.
|              arg1: Reference to the ziDAQRecorder 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( (ziDAQRecorder)arg1) -> object :
|          Reports the progress of the measurement with a number between
|          0 and 1.
|
|  read(...)
|      read( (ziDAQRecorder)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( (ziDAQRecorder)arg1) -> object
|
|  save(...)
|      save( (ziDAQRecorder)arg1, (str)arg2) -> None :
|          Save trigger data to file.
|              arg1: Reference to the ziDAQRecorder class.
|              arg2: File name string (without extension).
|
|  set(...)
|      set( (ziDAQRecorder)arg1, (str)arg2, (float)arg3) -> None :
|          Trigger 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.
|          trigger/device          string The device ID to execute the software
|                                         trigger, e.g. dev123
|                                         DEPRECATED - use absolute path in
trigger/0/triggernode
|          trigger/endless         bool   Enable endless triggering:
|                                         1 = enable,
|                                         0 = disable.
|          trigger/forcetrigger    bool   Force a trigger.
|          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 = hardware trigger on trigger line
source
|                                         6 = tracking edge trigger on source
|                                         7 = 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.
|          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/hwtrigsource  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/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/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/filename        string This parameter is deprecated. If
 |                                         specified, i.e. not empty, it enables
 |                                         automatic saving of data in single
 |                                         trigger mode (trigger/endless = 0).
 |          trigger/directory       string The directory where files are saved.
 |          trigger/fileformat      string The format of the file for saving data:
 |                                         0 = Matlab,
 |                                         1 = CSV,
 |                                         2 = ZView (Impedance data only).
 |          trigger/historylength   bool   Maximum number of entries stored in the
 |                                         measurement history.
 |          trigger/clearhistory    bool   Remove all records from the history list.
 |
 |      set( (ziDAQRecorder)arg1, (str)arg2, (int)arg3) -> None
 |
 |      set( (ziDAQRecorder)arg1, (str)arg2, (str)arg3) -> None
 |
 |      set( (ziDAQRecorder)arg1, (object)arg2) -> None :
 |              arg1: Reference to the ziDAQRecorder class.
 |              arg2: A list of path/value pairs.
 |
 |  subscribe(...)
 |      subscribe( (ziDAQRecorder)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 ziDAQRecorder 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( (ziDAQRecorder)arg1) -> None :
 |          Execute a manual trigger.
 |
 |  unsubscribe(...)
 |      unsubscribe( (ziDAQRecorder)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 ziDAQRecorder class.
 |              arg2: Path string of the node. Use wild card to
 |                    select all. Alternatively also a list of path
 |                    strings can be specified.
 |
 |  ----------------------------------------------------------------------
 |  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__
```

4.4.8. Help for ziPython's **ziPllAdvisor** class

An instance of `ziPllAdvisor` is initialized using the `pllAdvisor` method from `ziDAQServer`:

```
>>> help('zhinst.ziPython.ziDAQServer.pllAdvisor')

Help on built-in function pllAdvisor in zhinst.ziPython.ziDAQServer:

zhinst.ziPython.ziDAQServer.pllAdvisor = pllAdvisor(...)
    pllAdvisor( (ziDAQServer)arg1) -> ziPllAdvisor :
        Create a pllAdvisor class. This will start a thread for running an
        asynchronous pllAdvisor.
            arg1: Reference to the ziDAQServer class.
            arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED, ignored

    pllAdvisor( (ziDAQServer)arg1, (int)arg2) -> ziPllAdvisor
```

Reference help for the `ziPllAdvisor` class.

```
>>> help('zhinst.ziPython.ziPllAdvisor')

Help on class ziPllAdvisor in zhinst.ziPython:

zhinst.ziPython.ziPllAdvisor = class ziPllAdvisor(Boost.Python.instance)
 |  Method resolution order:
 |      ziPllAdvisor
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziPllAdvisor)arg1) -> None :
 |          End the pllAdvisor thread.
 |
 |  execute(...)
 |      execute( (ziPllAdvisor)arg1) -> None :
 |          Starts the pllAdvisor if not yet running.
 |
 |  finish(...)
 |      finish( (ziPllAdvisor)arg1) -> None :
 |          Stop the pllAdvisor.
 |
 |  finished(...)
 |      finished( (ziPllAdvisor)arg1) -> bool :
 |          Check if the command execution has finished. Returns True if finished.
 |
 |  get(...)
 |      get( (ziPllAdvisor)arg1, (str)arg2, (bool)arg3) -> object :
 |          Return a dict with all nodes from the specified sub-tree.
 |              arg1: Reference to the ziPllAdvisor 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( (ziPllAdvisor)arg1, (str)arg2) -> object
 |
 |  listNodes(...)
 |      listNodes( (ziPllAdvisor)arg1, (str)arg2, (int)arg3) -> list :
 |          This function returns a list of node names found at the specified path.
```

```
 |              arg1: Reference to the ziPllAdvisor 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( (ziPllAdvisor)arg1) -> object :
 |          Reports the progress of the command with a number between
 |          0 and 1.
 |
 |  read(...)
 |      read( (ziPllAdvisor)arg1, (bool)arg2) -> object :
 |          Read pllAdvisor 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( (ziPllAdvisor)arg1) -> object
 |
 |  save(...)
 |      save( (ziPllAdvisor)arg1, (str)arg2) -> None :
 |          Save PLL advisor data to file.
 |              arg1: Reference to the ziPllAdvisor class.
 |              arg2: File name string (without extension).
 |
 |  set(...)
 |      set( (ziPllAdvisor)arg1, (str)arg2, (float)arg3) -> None :
 |          PLL Advisor Parameters
 |          Path name              Type    Description
 |          pllAdvisor/bode        struct  Output parameter. Contains the resulting
 |                                         bode plot of the PLL simulation.
 |          pllAdvisor/calculate   int     Command to calculate values. Set to 1 to
 |                                         start the calculation.
 |          pllAdvisor/center      double  Center frequency of the PLL oscillator.
 |                                         The PLL frequency shift is relative to
 |                                         this center frequency.
 |          pllAdvisor/d           double  Differential gain.
 |          pllAdvisor/demodbw     double  Demodulator bandwidth used for the PLL
 |                                         loop filter
 |          pllAdvisor/i           double  Integral gain.
 |          pllAdvisor/mode        double  Select PLL Advisor mode. Currently only
 |                                         one mode (open loop) is supported.
 |          pllAdvisor/order       double  Demodulator order used for the PLL loop
 |                                         filter.
 |          pllAdvisor/p           double  Proportional gain.
 |          pllAdvisor/pllbw       double  Demodulator bandwidth used for the PLL
 |                                         loop filter.
 |          pllAdvisor/pm          double  Output parameter. Simulated phase margin
 |                                         of the PLL with the current settings. The
 |                                         phase margin should be greater than 45 deg
```

```
 |                                            and preferably greater than 65 deg for
 |                                            stable conditions.
 |           pllAdvisor/pmfreq     double  Output parameter. Simulated phase margin
 |                                            frequency.
 |           pllAdvisor/q          double  Quality factor. Currently not used.
 |           pllAdvisor/rate       double  PLL Advisor sampling rate of the PLL
 |                                            control loop.
 |           pllAdvisor/stable     int     Output parameter. When 1, the PLL Advisor
 |                                            found a stable solution with the given
 |                                            settings. When 0, revise your settings and
 |                                            rerun the PLL Advisor.
 |           pllAdvisor/targetbw   int     Requested PLL bandwidth. Higher
 |                                            frequencies may need manual tuning.
 |           pllAdvisor/targetfail int     Output parameter. 1 indicates the
 |                                            simulated PLL BW is smaller than the
 |                                            Target BW.
 |
 |       set( (ziPllAdvisor)arg1, (str)arg2, (int)arg3) -> None
 |
 |       set( (ziPllAdvisor)arg1, (str)arg2, (str)arg3) -> None
 |
 |       set( (ziPllAdvisor)arg1, (object)arg2) -> None :
 |               arg1: Reference to the ziPllAdvisor class.
 |               arg2: A list of path/value pairs.
 |
 |   subscribe(...)
 |       subscribe( (ziPllAdvisor)arg1, (str)arg2) -> None :
 |           Subscribe to one or several nodes.
 |
 |   trigger(...)
 |       trigger( (ziPllAdvisor)arg1) -> None :
 |           Not applicable to this module.
 |
 |   unsubscribe(...)
 |       unsubscribe( (ziPllAdvisor)arg1, (str)arg2) -> None :
 |           Unsubscribe from one or several nodes.
 |
 |   ----------------------------------------------------------------------
 |   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__
```

4.4.9. Help for ziPython's **ziPidAdvisor** class

An instance of `ziPidAdvisor` 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) -> ziPidAdvisor :
        Create a pidAdvisor class. This will start a thread for running an
        asynchronous pidAdvisor.
            arg1: Reference to the ziDAQServer class.
            arg2: Timeout in [ms]. Recommended value is 500ms. - DEPRECATED, ignored
```

```
    pidAdvisor( (ziDAQServer)arg1, (int)arg2) -> ziPidAdvisor
```

Reference help for the `ziPidAdvisor` class.

```
>>> help('zhinst.ziPython.ziPidAdvisor')

Help on class ziPidAdvisor in zhinst.ziPython:

zhinst.ziPython.ziPidAdvisor = class ziPidAdvisor(Boost.Python.instance)
 |  Method resolution order:
 |      ziPidAdvisor
 |      Boost.Python.instance
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      Raises an exception
 |      This class cannot be instantiated from Python
 |
 |  __reduce__ = <unnamed Boost.Python function>(...)
 |
 |  clear(...)
 |      clear( (ziPidAdvisor)arg1) -> None :
 |          End the pidAdvisor thread.
 |
 |  execute(...)
 |      execute( (ziPidAdvisor)arg1) -> None :
 |          Starts the pidAdvisor if not yet running.
 |
 |  finish(...)
 |      finish( (ziPidAdvisor)arg1) -> None :
 |          Stop the pidAdvisor.
 |
 |  finished(...)
 |      finished( (ziPidAdvisor)arg1) -> bool :
 |          Check if the command execution has finished. Returns True if finished.
 |
 |  get(...)
 |      get( (ziPidAdvisor)arg1, (str)arg2, (bool)arg3) -> object :
 |          Return a dict with all nodes from the specified sub-tree.
 |              arg1: Reference to the ziPidAdvisor 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( (ziPidAdvisor)arg1, (str)arg2) -> object
 |
 |  listNodes(...)
 |      listNodes( (ziPidAdvisor)arg1, (str)arg2, (int)arg3) -> list :
 |          This function returns a list of node names found at the specified path.
 |              arg1: Reference to the ziPidAdvisor 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( (ziPidAdvisor)arg1) -> object :
 |          Reports the progress of the command with a number between
 |          0 and 1.
 |
 |  read(...)
 |      read( (ziPidAdvisor)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( (ziPidAdvisor)arg1) -> object
 |
 |  save(...)
 |      save( (ziPidAdvisor)arg1, (str)arg2) -> None :
 |          Save PID advisor data to file.
 |              arg1: Reference to the ziPidAdvisor class.
 |              arg2: File name string (without extension).
 |
 |  set(...)
 |      set( (ziPidAdvisor)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.
 |          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/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.
 |
 |
 |      set( (ziPidAdvisor)arg1, (str)arg2, (int)arg3) -> None
 |
 |      set( (ziPidAdvisor)arg1, (str)arg2, (str)arg3) -> None
 |
 |      set( (ziPidAdvisor)arg1, (object)arg2) -> None :
 |              arg1: Reference to the ziPidAdvisor class.
 |              arg2: A list of path/value pairs.
```

```
|
|  subscribe(...)
|      subscribe( (ziPidAdvisor)arg1, (str)arg2) -> None :
|          Subscribe to one or several nodes.
|
|  trigger(...)
|      trigger( (ziPidAdvisor)arg1) -> None :
|          Not applicable to this module.
|
|  unsubscribe(...)
|      unsubscribe( (ziPidAdvisor)arg1, (str)arg2) -> None :
|          Unsubscribe from one or several nodes.
|
|  ----------------------------------------------------------------------
|  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 5. 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 5.1 for help Installing the LabOne LabVIEW API.
- Section 5.2 for help Getting Started with the LabOne LabVIEW API and running the examples.
- Section 5.3 for some LabVIEW Programming Tips and Tricks.

Note

This section and the provided examples are no substitute for a general LabVIEW tutorial. See, for example, the National Instruments website for help to get started programming with LabVIEW.

5.1. Installing the LabOne LabVIEW API

5.1.1. Requirements

A LabVIEW 2009 (or higher) installation is required on either Windows or Linux in order to use the LabOne LabVIEW API.

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.

5.1.2. 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 5.1.4.

5.1.3. Linux 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 at:

```
/usr/local/natinst/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 frmo the main LabOne tarball, the `Zurich Instruments LabOne/` directory is located in

   ```
   [PATH]/LabOneLinux64/API/LabVIEW/
   ```

3. Restart LabVIEW and verify your installation as described in Section 5.1.4.

5.1.4. Verifying your Installation

If the LabOne LabVIEW API palette can be accessed from within LabVIEW, the LabOne LabVIEW API is correctly installed. See Section 5.2.1 for help finding the palette.

5.2. Getting Started with the LabOne LabVIEW API

5.2.1. 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 5.1.

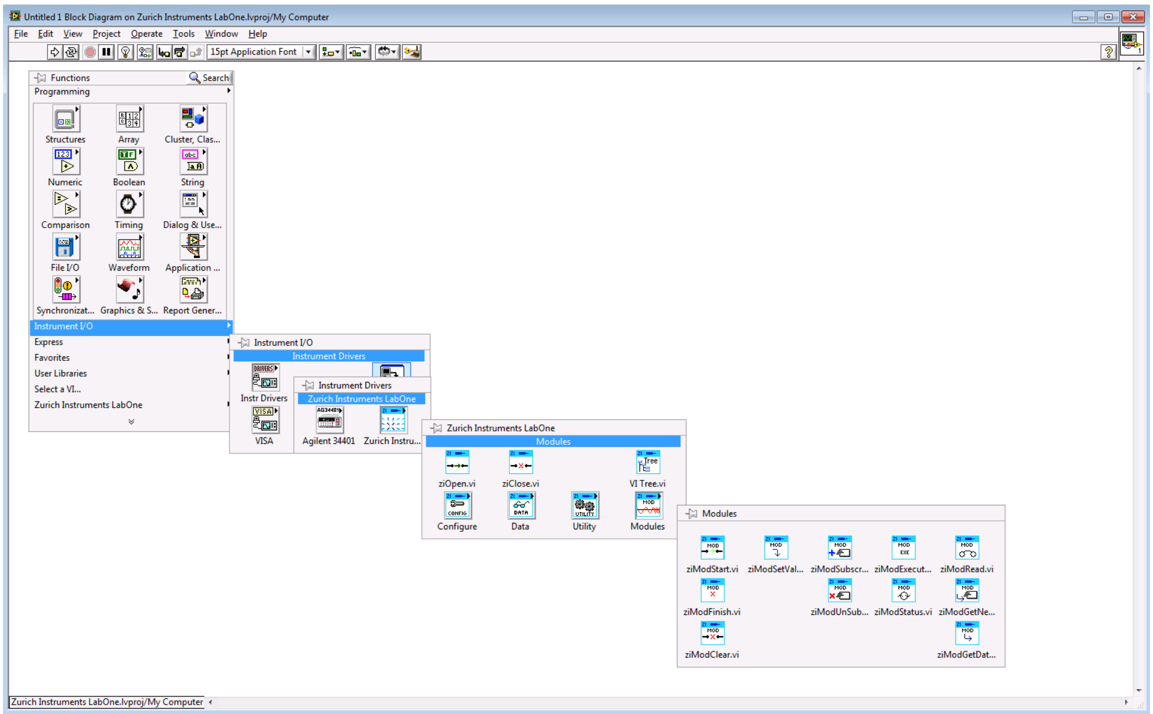

Figure 5.1. Locating the LabOne LabVIEW Palette

5.2.2. LabOne LabVIEW Programming Concepts

As described in Section 1.1 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 5.2. The `VI Tree.vi` can be found either via the LabOne VI palette, see Section 5.2.1, 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
```

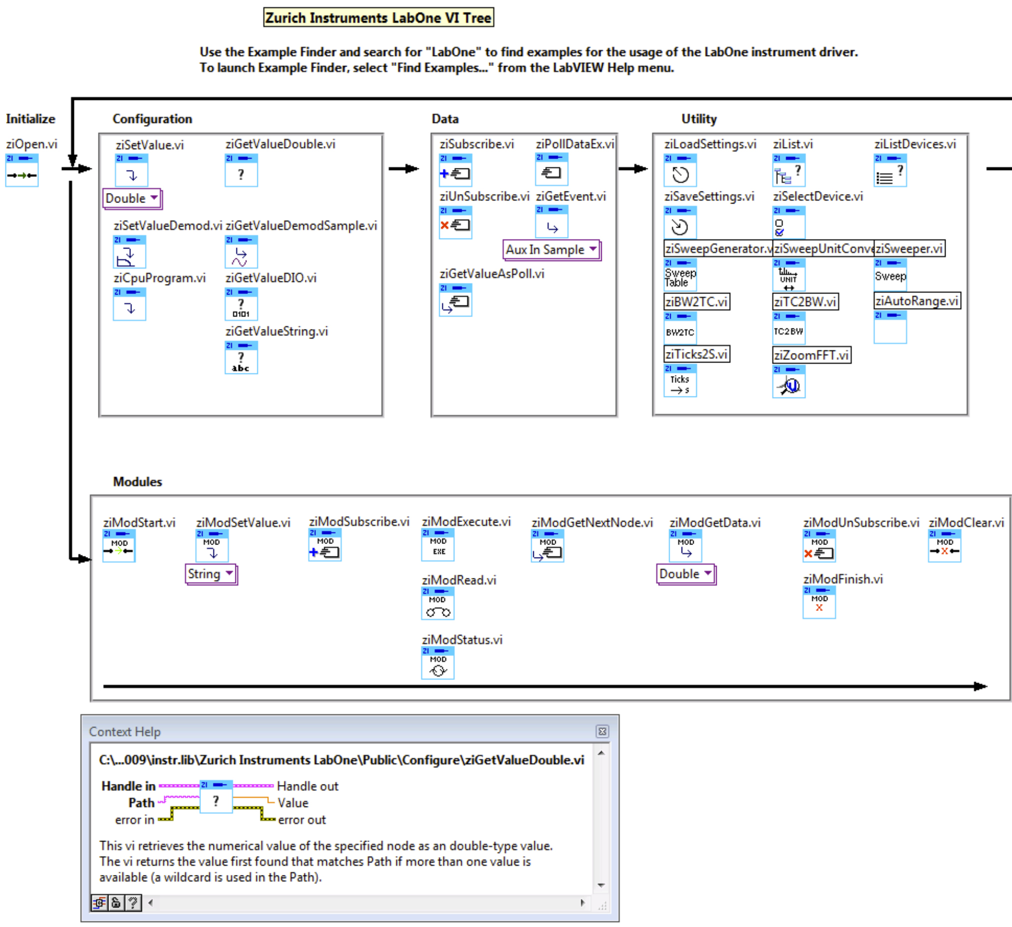

Figure 5.2. An overview of the LabOne LabVIEW VIs is given in `VI Tree.vi`. Press CTRL-h after selecting one of the VIs to obtain help.

5.2.3. Using ziCore Modules in the LabOne LabVIEW API

LabOne `ziCore` Modules 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`.

5.2.4. 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 5.2 for an example.

5.2.5. 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 5.3.

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.

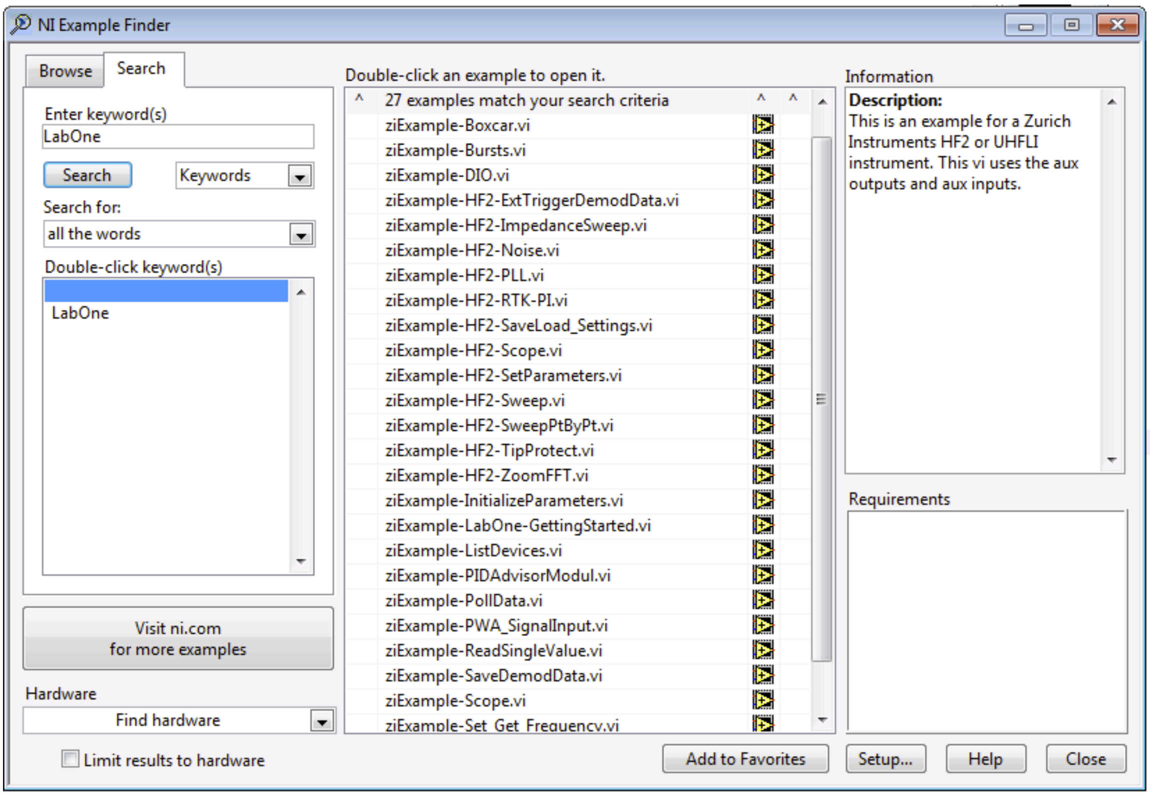

Figure 5.3. Search for "LabOne" in NI's Example Finder to find examples to run with your instrument.

5.2.6. 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 5.2.5) 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 5.4. 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.

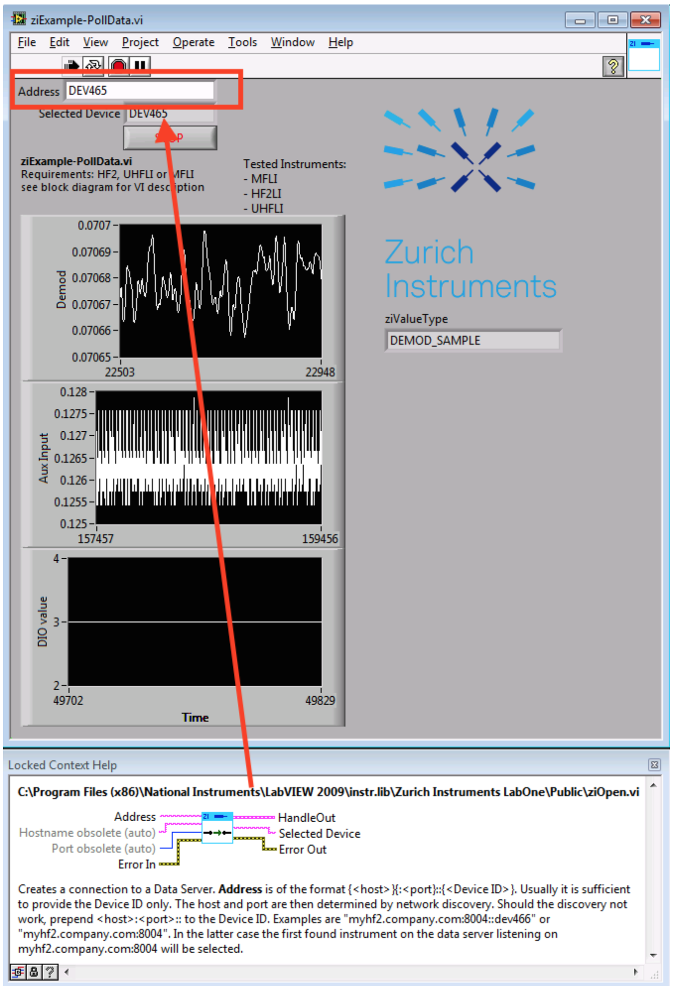

Figure 5.4. LabOne LabVIEW Example Poll Data: Device selection.

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 5.5.

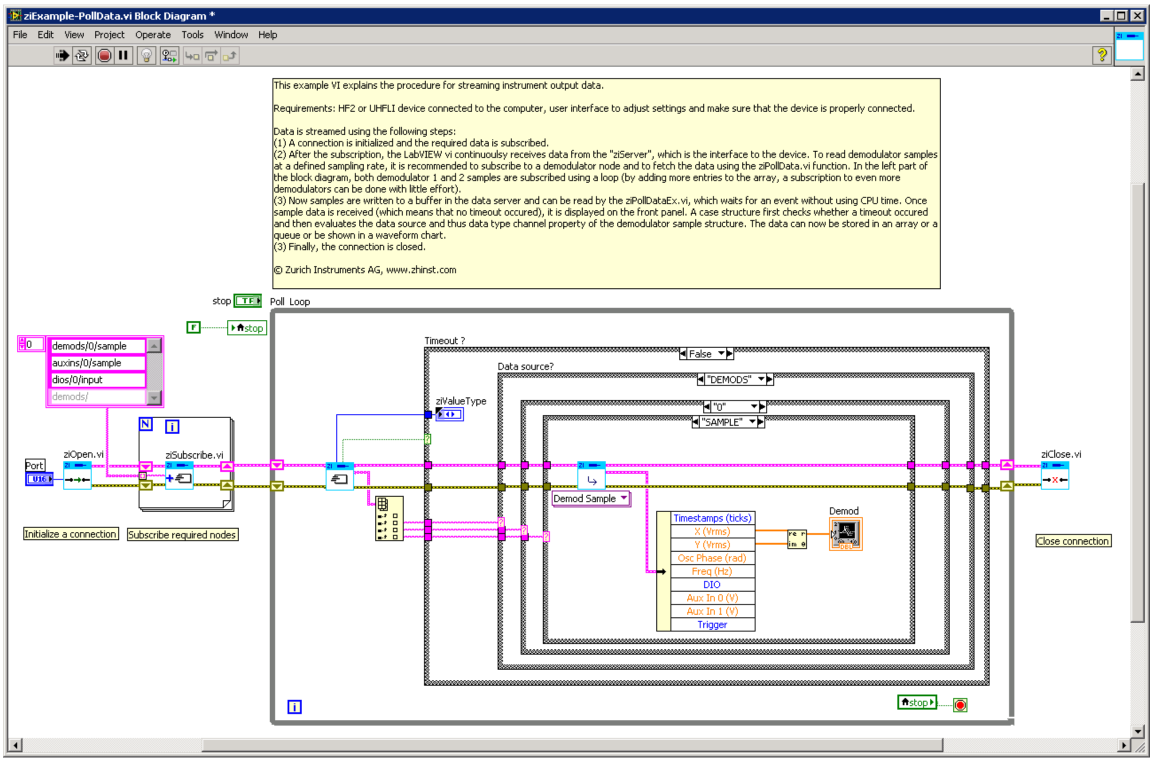

Figure 5.5. LabOne LabVIEW Example Poll Data: Block Diagram.

5.3. 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 1.1. 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.

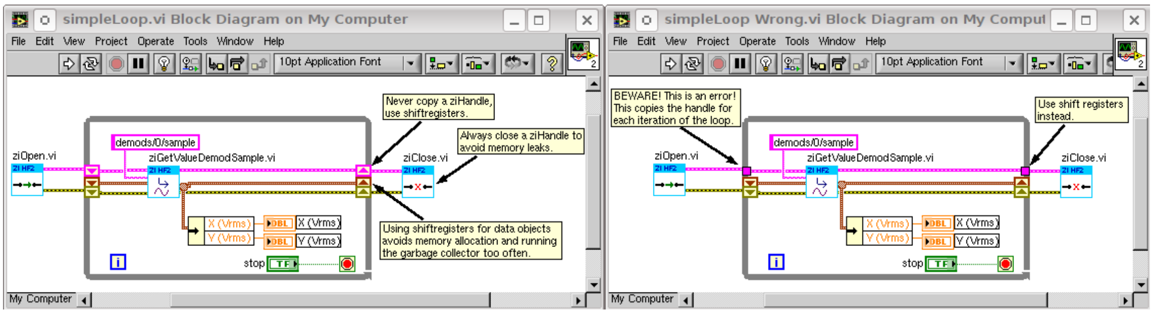

Figure 5.6. Examples of simple LabVIEW programs for the Zurich Instruments HF2 Series. Left: A well implemented loop, Right: An example for-loop gone wrong.

Chapter 6. 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.

6.1. 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.

6.1.1. Example

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;
  char* errBuffer;

  ZIConnection conn;

  // Initialize ZIConnection.
  if ((retVal = ziAPIInit(&conn)) != 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.
  char serverAddress[] = "localhost";
  if ((retVal = ziAPIConnectEx(conn, serverAddress, 8004, ZI_API_VERSION_5, NULL)) !
= 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 all devices to 150 Hz
    if ((retVal = ziAPISetValueD(conn,
                                 "/dev1046/demods/*/rate",
                                 150)) != 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;
```

```
    }
```

6.2. Module Documentation

6.2.1. 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 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 }

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 ziAPIGetRevision (unsigned int* revision)

  Retrieves the revision of ziAPI.

Detailed Description

```
// Copyright [2016] Zurich Instruments AG
#include <stdio.h>

#include "ziAPI.h"

int main() {
  ZIResult_enum retVal;
  ZIConnection conn;
  char* errBuffer;

  // Initialize ZIConnection.
  if ((retVal = ziAPIInit(&conn)) != 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.
  char serverAddress[] = "localhost";
  if ((retVal = ziAPIConnectEx(conn, serverAddress, 8004, ZI_API_VERSION_5, NULL)) !
= 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

enum ZIAPIVersion_enum

Enumerator:

- ZI_API_VERSION_0
- ZI_API_VERSION_1
- ZI_API_VERSION_4
- ZI_API_VERSION_5

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

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.

6.2.2. 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_LEAFSONLY, ZI_LIST_NODES_SETTINGSONLY, ZI_LIST_NONE, ZI_LIST_RECURSIVE, ZI_LIST_ABSOLUTE, ZI_LIST_LEAFSONLY, ZI_LIST_SETTINGSONLY }

  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 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.

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

enum ZIListNodes_enum

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_NONE

  Default, return a simple listing of the given node immediate descendants.

- ZI_LIST_RECURSIVE

  List the nodes recursively.

- ZI_LIST_ABSOLUTE

  Return absolute paths.

- ZI_LIST_LEAFSONLY

  Return only leaf nodes, which means the nodes at the outermost level of the tree.

- ZI_LIST_SETTINGSONLY

  Return only nodes which are marked as setting.

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 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

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

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

6.2.3. 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 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 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 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 ore 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 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 [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIDoubleData ValueD;

  if ((RetVal = ziAPISetValueI(Conn,
                               "/dev1046/demods/*/rate",
                               100)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
  }
```

```
  if ((RetVal = ziAPIGetValueD(Conn,
                               "/dev1046/demods/0/rate",
                               &ValueD)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
  } else {
    printf("Value = %f\n", ValueD);
  }
}
```

See Also:

ziAPISetValueD, 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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIIntegerData ValueI;

  if ((RetVal = ziAPISetValueD(Conn,
                               "/dev1046/demods/*/rate",
                               5.53)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
  }

  if ((RetVal = ziAPIGetValueI(Conn,
                               "/dev1046/demods/0/rate",
```

```
                                  &ValueI)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
  } else {
    printf("Value = %f\n", (float)ValueI);
  }
}
```

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 GetSample(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 GetSample(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 GetSample(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 doesn't reach 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

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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIIntegerData ValueI;

  if ((RetVal = ziAPISetValueD(Conn,
                               "/dev1046/demods/*/rate",
                               5.53)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
```

```
    }

    if ((RetVal = ziAPIGetValueI(Conn,
                                 "/dev1046/demods/0/rate",
                                 &ValueI)) != ZI_INFO_SUCCESS) {
      ziAPIGetError(RetVal, &ErrBuffer, NULL);
      fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
      printf("Value = %f\n", (float)ValueI);
    }
  }
```

See Also:

ziAPIGetValueD, ziAPISyncSetValueD

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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIDoubleData ValueD;

  if ((RetVal = ziAPISetValueI(Conn,
                               "/dev1046/demods/*/rate",
                               100)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
```

```
    }

    if ((RetVal = ziAPIGetValueD(Conn,
                                 "/dev1046/demods/0/rate",
                                 &ValueD)) != ZI_INFO_SUCCESS) {
      ziAPIGetError(RetVal, &ErrBuffer, NULL);
      fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
      printf("Value = %f\n", ValueD);
    }
  }
```

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 ore 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

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 ore 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

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)

6.2.4. 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;
  ZIByteArray* byteArray;
  ZIByteArrayTS* byteArrayTS;
  ZICntSample* cntSample;
  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

- ZIIntegerData* integerData
  when valueType == ZI_VALUE_TYPE_INTEGER_DATA
- ZIIntegerDataTS* integerDataTS
  when valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS
- ZIByteArray* byteArray
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY
- ZIByteArrayTS* byteArrayTS
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS
- ZICntSample* cntSample
  when valueType == ZI_VALUE_TYPE_CNT_SAMPLE
- ZITreeChangeData* 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
- ZIAuxInSample* 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

6.2.5. 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, ZIIntegerData value)

- ZIResult_enum ziAPIAsyncSetByteArray (ZIConnection conn, const char* path, uint8_t* buffer, uint32_t length)

- 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)

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)

6.2.6. 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

6.2.7. 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 ZISWTriggerHeader

  Structure to hold information about data returned from the SW Trigger Module.

- struct ZIModuleHeaderSweeper

  Structure to hold information about data returned from the Sweep Module.

- struct ZIModuleHeader

  Module-specific event header.

- 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 ZIModuleHeaderType_enum
  { ZI_MODULE_HEADER_TYPE_NONE,
  ZI_MODULE_HEADER_TYPE_SWTRIGGER,
  ZI_MODULE_HEADER_TYPE_SWEEPER }

  Enumerates all module header types.

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 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.

Data Structure Documentation

struct ZISWTriggerHeader

Structure to hold information about data returned from the SW Trigger Module.

```
#include "ziAPI.h"
```

```
typedef struct ZISWTriggerHeader {
  ZITimeStamp triggerStart;
  uint64_t triggerNumber;
  uint32_t cols;
  uint32_t rows;
  uint32_t repetitions;
  uint32_t flags;
  uint8_t reserved0[32];
} ZISWTriggerHeader;
```

Data Fields

- ZITimeStamp triggerStart

  Trigger timestamp.

- uint64_t triggerNumber

  Trigger counter since execution start.

- uint32_t cols

  Number of columns.

- uint32_t rows

  Number of rows.

- uint32_t repetitions

  Number of repetitions in grid mode.

- uint32_t flags

  Flags Bit (0): Finished (all repetitions recorded)

- uint8_t reserved0

  Reserved space for future use.

struct ZIModuleHeaderSweeper

Structure to hold information about data returned from the Sweep Module.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeaderSweeper {
  char traceName[256];
} ZIModuleHeaderSweeper;
```

Data Fields

- char traceName

struct ZIModuleHeader

Module-specific event header.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeader {
  ZIModuleHeaderType_enum
                    type;
  void* untyped;
  ZISWTriggerHeader* swTrigger;
  ZISweeperHeader* sweeper;
  union ZIModuleHeader::@7 ptr;
} ZIModuleHeader;
```

Data Fields

- ZIModuleHeaderType_enum type
- void* untyped
- ZISWTriggerHeader* swTrigger
- ZISweeperHeader* sweeper
- union ZIModuleHeader::@7 ptr

struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleEvent {
  uint64_t allocatedSize;
  ZIModuleHeader
                  header;
  ZIEvent
                  value[0];
} ZIModuleEvent;
```

Data Fields

- uint64_t allocatedSize

  For internal use - never modify!

- ZIModuleHeader header

  Module-specific event header.

- ZIEvent value

  Defines location of stored ZIEvent.

Enumeration Type Documentation

enum ZIModuleHeaderType_enum

Enumerates all module header types.

Enumerator:

- ZI_MODULE_HEADER_TYPE_NONE
- ZI_MODULE_HEADER_TYPE_SWTRIGGER
- ZI_MODULE_HEADER_TYPE_SWEEPER

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.

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

ziAPIModSetIntegerData

ZIResult_enum ziAPIModSetIntegerData (ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData 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

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

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 specifyin 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 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 callin 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 ppreallocates 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

6.2.8. 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

enum ZIVectorWriteStatus_enum

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

6.2.9. Device discovery

Functions for working with device discovery.

Functions

- ZIResult_enum ziAPIDiscoveryFind (ZIConnection conn, const char* deviceAddress, const char** deviceId)

  Returns the device id for a given device address.
  Attention! Invalidates all pointers previously returned by ziAPIDiscovery* calls.

- ZIResult_enum ziAPIDiscoveryGet (ZIConnection conn, const char* deviceId, const char** propsJSON)

  Returns the device properties for a given device id in JSON format.

- ZIResult_enum ziAPIDiscoveryGetValueI (ZIConnection conn, const char* deviceId, const char* propName, ZIIntegerData* value)

  Returns given integer property value for a given device id.

- ZIResult_enum ziAPIDiscoveryGetValueS (ZIConnection conn, const char* deviceId, const char* propName, const char** value)

  Returns given string property value for a given device id.

Function Documentation

ziAPIDiscoveryFind

ZIResult_enum ziAPIDiscoveryFind (ZIConnection conn, const char* deviceAddress, const char** deviceId)

Returns the device id for a given device address. Attention! Invalidates all pointers previously returned by ziAPIDiscovery* calls.

ziAPIDiscoveryGet

ZIResult_enum ziAPIDiscoveryGet (ZIConnection conn, const char* deviceId, const char propsJSON)**

Returns the device properties for a given device id in JSON format.

ziAPIDiscoveryGetValueI

ZIResult_enum ziAPIDiscoveryGetValueI (ZIConnection conn, const char* deviceId, const char* propName, ZIIntegerData* value)

Returns given integer property value for a given device id.

ziAPIDiscoveryGetValueS

ZIResult_enum ziAPIDiscoveryGetValueS (ZIConnection conn, const char* deviceId, const char* propName, const char** value)

Returns given string property value for a given device id.

6.3. Data Structure Documentation

6.3.1. 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

6.3.2. 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

6.3.3. 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

6.3.4. 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

6.3.5. 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.

6.3.6. 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.

6.3.7. 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

6.3.8. 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.

6.3.9. struct ZIAdvisorSample

```
typedef struct ZIAdvisorSample {
  double grid;
  double x;
  double y;
} ZIAdvisorSample;
```

Data Fields

- double grid

  Grid.

- double x

  X.

- double y

  Y.

6.3.10. 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.

6.3.11. 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.

6.3.12. 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.

6.3.13. 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.

6.3.14. 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

6.3.15. struct ZICntSample

The structure used to hold data for a single counter sample.

```
#include "ziAPI.h"
```

```
typedef struct ZICntSample {
  ZITimeStamp timeStamp;
  uint16_t counter;
  uint16_t reserved0;
  uint16_t id;
  uint16_t reserved1;
} ZICntSample;
```

Data Fields

- ZITimeStamp timeStamp
  The timestamp at which the values have been measured.
- uint16_t counter
  Counter value.
- uint16_t reserved0
  Reserved.
- uint16_t id
  Trigger id.
- uint16_t reserved1
  Reserved.

6.3.16. 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
  Frequency at that sample.
- double phase
  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.

6.3.17. 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.

6.3.18. 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

6.3.19. 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;
  ZIByteArray* byteArray;
  ZIByteArrayTS* byteArrayTS;
  ZICntSample* cntSample;
  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

- ZIIntegerData* integerData

when valueType == ZI_VALUE_TYPE_INTEGER_DATA

- ZIIntegerDataTS* integerDataTS
  when valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS
- ZIByteArray* byteArray
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY
- ZIByteArrayTS* byteArrayTS
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS
- ZICntSample* cntSample
  when valueType == ZI_VALUE_TYPE_CNT_SAMPLE
- ZITreeChangeData* 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
- ZIAuxInSample* 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

6.3.20. 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

6.3.21. 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.

6.3.22. 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

6.3.23. struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleEvent {
  uint64_t allocatedSize;
  ZIModuleHeader
            header;
  ZIEvent
            value[0];
} ZIModuleEvent;
```

Data Fields

- uint64_t allocatedSize

  For internal use - never modify!

- ZIModuleHeader header

  Module-specific event header.

- ZIEvent value

  Defines location of stored ZIEvent.

6.3.24. struct ZIModuleHeader

Module-specific event header.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeader {
  ZIModuleHeaderType_enum
            type;
  void* untyped;
  ZISWTriggerHeader* swTrigger;
  ZISweeperHeader* sweeper;
  union ZIModuleHeader::@7 ptr;
} ZIModuleHeader;
```

Data Fields

- ZIModuleHeaderType_enum type
- void* untyped
- ZISWTriggerHeader* swTrigger
- ZISweeperHeader* sweeper
- union ZIModuleHeader::@7 ptr

6.3.25. struct ZIModuleHeaderSweeper

Structure to hold information about data returned from the Sweep Module.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeaderSweeper {
  char traceName[256];
} ZIModuleHeaderSweeper;
```

Data Fields

- char traceName

6.3.26. 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.

6.3.27. 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.

6.3.28. struct ZIScopeWave

The structure used to hold scope data. 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

  Time stamp of the last sample in this data block.

- ZITimeStamp triggerTimeStamp

  Time stamp 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: 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(0): rising edge trigger off = 0, on = 1. Bit(1): falling edge trigger off = 0, on = 1.

- uint8_t triggerInput

  Trigger source (same values as for channel input)

- uint8_t reserved0

- uint8_t channelBWLimit

  Bandwidth-limit flag, per channel. Bit(0): off = 0, on = 1 Bit(7...1): Reserved.

- uint8_t channelMath

  Math Operation (e.g averaging) Bit (7..0): 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 SingleTransfer = 0, BlockTransfer = 1, ContinuousTransfer = 3, FFTSingleTransfer = 4. Other values are reserved.

- uint8_t blockMarker

  Block marker: Bit (0): 1 = End marker for continuous or multi-block transfer Bit (7..0): Reserved.

- uint8_t flags

  Indicator Flags. 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: Int16 = 0, Int32 = 1, Float = 2, Int16Interleaved = 4, Int32Interleaved = 5, FloatInterleaved = 6.

- uint32_t sampleCount

  Number of samples in one channel in the current block, same for all channels.

- int16_t dataInt16

- int32_t dataInt32

- float dataFloat

- union ZIScopeWave::@0 data
  First wave data.

6.3.29. struct ZIScopeWaveEx

The structure used to hold scope data (extended). 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];
  uint64_t reserved1[32];
  int16_t dataInt16[0];
  int32_t dataInt32[0];
  float dataFloat[0];
  union ZIScopeWaveEx::@1 data;
} ZIScopeWaveEx;
```

Data Fields

- ZITimeStamp timeStamp

  Time stamp of the last sample in this data block.

- ZITimeStamp triggerTimeStamp

  Time stamp 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: 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(0): rising edge trigger off = 0, on = 1. Bit(1): falling edge trigger off = 0, on = 1.

- uint8_t triggerInput

  Trigger source (same values as for channel input)

- uint8_t reserved0

- uint8_t channelBWLimit

  Bandwidth-limit flag, per channel. Bit(0): off = 0, on = 1 Bit(7…1): Reserved.

- uint8_t channelMath

  Math Operation (e.g averaging) Bit (7..0): 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 SingleTransfer = 0, BlockTransfer = 1, ContinuousTransfer = 3, FFTSingleTransfer = 4. Other values are reserved.

- uint8_t blockMarker

  Block marker: Bit (0): 1 = End marker for continuous or multi-block transfer Bit (7..0): Reserved.

- uint8_t flags

  Indicator Flags. 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: Int16 = 0, Int32 = 1, Float = 2, Int16Interleaved = 4, Int32Interleaved = 5, FloatInterleaved = 6.

- 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.
- uint64_t reserved1
- int16_t dataInt16
- int32_t dataInt32
- float dataFloat
- union ZIScopeWaveEx::@1 data
  First wave data.

6.3.30. 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.

6.3.31. 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.

- 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[.

6.3.32. 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.

6.3.33. 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.

6.3.34. 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.

6.3.35. 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)

6.3.36. 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.

- 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.

6.3.37. 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.

6.3.38. 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.

6.3.39. struct ZISWTriggerHeader

Structure to hold information about data returned from the SW Trigger Module.

```
#include "ziAPI.h"
```

```
typedef struct ZISWTriggerHeader {
  ZITimeStamp triggerStart;
  uint64_t triggerNumber;
  uint32_t cols;
  uint32_t rows;
  uint32_t repetitions;
  uint32_t flags;
  uint8_t reserved0[32];
} ZISWTriggerHeader;
```

Data Fields

- ZITimeStamp triggerStart
  Trigger timestamp.
- uint64_t triggerNumber
  Trigger counter since execution start.
- uint32_t cols
  Number of columns.
- uint32_t rows
  Number of rows.
- uint32_t repetitions
  Number of repetitions in grid mode.
- uint32_t flags
  Flags Bit (0): Finished (all repetitions recorded)
- uint8_t reserved0
  Reserved space for future use.

6.3.40. 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.

6.3.41. 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.

6.4. File Documentation

6.4.1. File ziAPI.h

Header File for the LabOne C/C++ API.

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 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 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. 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). 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 ZIStatisticSample

- 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 ZISWTriggerHeader

  Structure to hold information about data returned from the SW Trigger Module.

- struct ZIModuleHeaderSweeper

  Structure to hold information about data returned from the Sweep Module.

- struct ZIModuleHeader

  Module-specific event header.

- 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 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_MAX, ZI_SUCCESS, ZI_MAX_INFO, ZI_WARNING, ZI_UNDERRUN, ZI_OVERFLOW, ZI_NOTFOUND, ZI_MAX_WARNING, ZI_ERROR, ZI_USB, ZI_MALLOC, ZI_MUTEX_INIT, ZI_MUTEX_DESTROY, ZI_MUTEX_LOCK, ZI_MUTEX_UNLOCK, ZI_THREAD_START, ZI_THREAD_JOIN, ZI_SOCKET_INIT,

ZI_SOCKET_CONNECT, ZI_HOSTNAME, ZI_CONNECTION, ZI_TIMEOUT, ZI_COMMAND, ZI_SERVER_INTERNAL, ZI_LENGTH, ZI_FILE, ZI_DUPLICATE, ZI_READONLY, ZI_MAX_ERROR }

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_SCOPE_WAVE, ZI_VALUE_TYPE_SCOPE_WAVE_EX, ZI_VALUE_TYPE_BYTE_ARRAY_TS, ZI_VALUE_TYPE_CNT_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, ZI_DATA_NONE, ZI_DATA_DOUBLE, ZI_DATA_INTEGER, ZI_DATA_DEMODSAMPLE, ZI_DATA_SCOPEWAVE, ZI_DATA_AUXINSAMPLE, ZI_DATA_DIOSAMPLE, ZI_DATA_BYTEARRAY, ZI_DATA_TREE_CHANGED }

  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_SUPPRESSION_PARAM0, ZI_IMP_FLAGS_SUPPRESSION_PARAM1, ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM0, ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM1, 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 }

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_ASCIIZ }

  Enumerates all the types that a ::elementType may have.

- enum ZIAPIVersion_enum { ZI_API_VERSION_0,
ZI_API_VERSION_1, ZI_API_VERSION_4, ZI_API_VERSION_5 }

- enum ZIListNodes_enum { ZI_LIST_NODES_NONE,
ZI_LIST_NODES_RECURSIVE, ZI_LIST_NODES_ABSOLUTE,
ZI_LIST_NODES_LEAFSONLY,
ZI_LIST_NODES_SETTINGSONLY, ZI_LIST_NONE,
ZI_LIST_RECURSIVE, ZI_LIST_ABSOLUTE,
ZI_LIST_LEAFSONLY, ZI_LIST_SETTINGSONLY }

  Defines the values of the flags used in ziAPIListNodes.

- enum ZIModuleHeaderType_enum
{ ZI_MODULE_HEADER_TYPE_NONE,
ZI_MODULE_HEADER_TYPE_SWTRIGGER,
ZI_MODULE_HEADER_TYPE_SWEEPER }

  Enumerates all module header types.

- 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 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 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 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 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 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 ore 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 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 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 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 ziAPIDiscoveryFind (ZIConnection conn, const char* deviceAddress, const char** deviceId)

  Returns the device id for a given device address.
  Attention! Invalidates all pointers previously returned by ziAPIDiscovery* calls.

- ZIResult_enum ziAPIDiscoveryGet (ZIConnection conn, const char* deviceId, const char** propsJSON)

  Returns the device properties for a given device id in JSON format.

- ZIResult_enum ziAPIDiscoveryGetValueI (ZIConnection conn, const char* deviceId, const char* propName, ZIIntegerData* value)

  Returns given integer property value for a given device id.

- ZIResult_enum ziAPIDiscoveryGetValueS (ZIConnection conn, const char* deviceId, const char* propName, const char** value)

  Returns given string property value for a given device id.

- __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 timeStamp;
  ZIDoubleData value;
} ZIDoubleDataTS;
```

Data Fields

- ZITimeStamp timeStamp
  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 timeStamp;
  ZIIntegerData value;
} ZIIntegerDataTS;
```

Data Fields

- ZITimeStamp timeStamp

  Time stamp at which the value has changed.

- ZIIntegerData value

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

  Frequency at that sample.

- double phase

  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;
  uint16_t counter;
  uint16_t reserved0;
  uint16_t id;
  uint16_t reserved1;
} ZICntSample;
```

Data Fields

- ZITimeStamp timeStamp
  The timestamp at which the values have been measured.
- uint16_t counter
  Counter value.
- uint16_t reserved0
  Reserved.
- uint16_t id
  Trigger id.
- uint16_t reserved1
  Reserved.

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. 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

  Time stamp of the last sample in this data block.

- ZITimeStamp triggerTimeStamp

  Time stamp 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: 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(0): rising edge trigger off = 0, on = 1. Bit(1): falling edge trigger off = 0, on = 1.

- uint8_t triggerInput

  Trigger source (same values as for channel input)

- uint8_t reserved0

- uint8_t channelBWLimit

  Bandwidth-limit flag, per channel. Bit(0): off = 0, on = 1 Bit(7…1): Reserved.

- uint8_t channelMath

  Math Operation (e.g averaging) Bit (7..0): 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 SingleTransfer = 0, BlockTransfer = 1, ContinuousTransfer = 3, FFTSingleTransfer = 4. Other values are reserved.

- uint8_t blockMarker

  Block marker: Bit (0): 1 = End marker for continuous or multi-block transfer Bit (7..0): Reserved.

- uint8_t flags

  Indicator Flags. 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: Int16 = 0, Int32 = 1, Float = 2, Int16Interleaved = 4, Int32Interleaved = 5, FloatInterleaved = 6.

- uint32_t sampleCount

  Number of samples in one channel in the current block, same for all channels.

- int16_t dataInt16

- int32_t dataInt32

- float dataFloat

- union ZIScopeWave::@0 data
  First wave data.

struct ZIScopeWaveEx

The structure used to hold scope data (extended). 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];
  uint64_t reserved1[32];
  int16_t dataInt16[0];
  int32_t dataInt32[0];
  float dataFloat[0];
  union ZIScopeWaveEx::@1 data;
} ZIScopeWaveEx;
```

Data Fields

- ZITimeStamp timeStamp

  Time stamp of the last sample in this data block.

- ZITimeStamp triggerTimeStamp

  Time stamp 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: 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(0): rising edge trigger off = 0, on = 1. Bit(1): falling edge trigger off = 0, on = 1.

- uint8_t triggerInput

  Trigger source (same values as for channel input)

- uint8_t reserved0

- uint8_t channelBWLimit

  Bandwidth-limit flag, per channel. Bit(0): off = 0, on = 1 Bit(7...1): Reserved.

- uint8_t channelMath

  Math Operation (e.g averaging) Bit (7..0): 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 SingleTransfer = 0, BlockTransfer = 1, ContinuousTransfer = 3, FFTSingleTransfer = 4. Other values are reserved.

- uint8_t blockMarker

  Block marker: Bit (0): 1 = End marker for continuous or multi-block transfer Bit (7..0): Reserved.

- uint8_t flags

  Indicator Flags. 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: Int16 = 0, Int32 = 1, Float = 2, Int16Interleaved = 4, Int32Interleaved = 5, FloatInterleaved = 6.

- 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.

- uint64_t reserved1

- int16_t dataInt16

- int32_t dataInt32

- float dataFloat

- union ZIScopeWaveEx::@1 data
  First wave data.

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.

- 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.

- 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;
  ZIIntegerData* integerData;
  ZIIntegerDataTS* integerDataTS;
  ZIByteArray* byteArray;
  ZIByteArrayTS* byteArrayTS;
  ZICntSample* cntSample;
  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

- ZIIntegerData* integerData

when valueType == ZI_VALUE_TYPE_INTEGER_DATA

- ZIIntegerDataTS* integerDataTS
  when valueType == ZI_VALUE_TYPE_INTEGER_DATA_TS
- ZIByteArray* byteArray
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY
- ZIByteArrayTS* byteArrayTS
  when valueType == ZI_VALUE_TYPE_BYTE_ARRAY_TS
- ZICntSample* cntSample
  when valueType == ZI_VALUE_TYPE_CNT_SAMPLE
- ZITreeChangeData* 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
- ZIAuxInSample* 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 ZISWTriggerHeader

Structure to hold information about data returned from the SW Trigger Module.

```
#include "ziAPI.h"
```

```
typedef struct ZISWTriggerHeader {
  ZITimeStamp triggerStart;
  uint64_t triggerNumber;
  uint32_t cols;
  uint32_t rows;
  uint32_t repetitions;
  uint32_t flags;
  uint8_t reserved0[32];
} ZISWTriggerHeader;
```

Data Fields

- ZITimeStamp triggerStart

  Trigger timestamp.

- uint64_t triggerNumber

  Trigger counter since execution start.

- uint32_t cols

  Number of columns.

- uint32_t rows

  Number of rows.

- uint32_t repetitions

  Number of repetitions in grid mode.

- uint32_t flags

  Flags Bit (0): Finished (all repetitions recorded)

- uint8_t reserved0

  Reserved space for future use.

struct ZIModuleHeaderSweeper

Structure to hold information about data returned from the Sweep Module.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeaderSweeper {
  char traceName[256];
} ZIModuleHeaderSweeper;
```

Data Fields

- char traceName

struct ZIModuleHeader

Module-specific event header.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleHeader {
  ZIModuleHeaderType_enum
                type;
  void* untyped;
  ZISWTriggerHeader* swTrigger;
  ZISweeperHeader* sweeper;
  union ZIModuleHeader::@7 ptr;
} ZIModuleHeader;
```

Data Fields

- ZIModuleHeaderType_enum type
- void* untyped
- ZISWTriggerHeader* swTrigger
- ZISweeperHeader* sweeper
- union ZIModuleHeader::@7 ptr

struct ZIModuleEvent

This struct holds data of a single chunk from module lookup.

```
#include "ziAPI.h"
```

```
typedef struct ZIModuleEvent {
  uint64_t allocatedSize;
  ZIModuleHeader
                  header;
  ZIEvent
                  value[0];
} ZIModuleEvent;
```

Data Fields

- uint64_t allocatedSize

  For internal use - never modify!

- ZIModuleHeader header

  Module-specific event 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

enum ZIResult_enum

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_MAX
- ZI_SUCCESS

  Success (no error)
- ZI_MAX_INFO
- ZI_WARNING

  Warning (general);.
- ZI_UNDERRUN

  FIFO Underrun.
- ZI_OVERFLOW

  FIFO Overflow.
- ZI_NOTFOUND

  Value or Node not found.
- ZI_MAX_WARNING
- ZI_ERROR

  Error (general)
- ZI_USB

  USB Communication failed.
- ZI_MALLOC

  Memory allocation failed.
- ZI_MUTEX_INIT

  Unable to initialize mutex.
- ZI_MUTEX_DESTROY

  Unable to destroy mutex.
- ZI_MUTEX_LOCK

  Unable to lock mutex.
- ZI_MUTEX_UNLOCK

  Unable to unlock mutex.

- ZI_THREAD_START
  Unable to start thread.
- ZI_THREAD_JOIN
  Unable to join thread.
- ZI_SOCKET_INIT
  Can't initialize socket.
- ZI_SOCKET_CONNECT
  Unable to connect socket.
- ZI_HOSTNAME
  Hostname not found.
- ZI_CONNECTION
  Connection invalid.
- ZI_TIMEOUT
  Command timed out.
- ZI_COMMAND
  Command internally failed.
- ZI_SERVER_INTERNAL
  Command failed in server.
- ZI_LENGTH
  Provided Buffer length doesn't reach.
- ZI_FILE
  Can't open file or read from it.
- ZI_DUPLICATE
  There is already a similar entry.
- ZI_READONLY
  Attempt to set a read-only node.
- ZI_MAX_ERROR

enum ZIValueType_enum

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_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_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.

- ZI_DATA_NONE

  no data type. the ziEvent is invalid.

- ZI_DATA_DOUBLE

  double data type. use the ziEvent::Val.Double Pointer to read the data of the event.

- ZI_DATA_INTEGER

  integer data type. use the ziEvent::Val.Integer Pointer to read the data of the event.

- ZI_DATA_DEMODSAMPLE

  DemodSample data type. use the ziEvent::Val.Sample Pointer to read the data of the event.

- ZI_DATA_SCOPEWAVE

  ScopeWave data type. use the ziEvent::Val.Wave Pointer to read the data of the event.

- ZI_DATA_AUXINSAMPLE

  MiscADValue data type. use the ziEvent::Val.ADValue Pointer to read the data of the event.

- ZI_DATA_DIOSAMPLE

  DIOValue data type. use the ziEvent::Val.DIOValue Pointer to read the data of the event.

- ZI_DATA_BYTEARRAY

  ByteArray data type. use the ziEvent::Val.ByteArray Pointer to read the data of the event.

- ZI_DATA_TREE_CHANGED

  a list of added or removed trees. use the ziEvent::Val.Tree Pointer to read the data of the event.

enum ZITreeAction_enum

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.

enum ZIImpFlags_enum

Enumerates the bits set in an ZIImpedanceSample's flags.

Enumerator:

- 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_SUPPRESSION_PARAM0
- ZI_IMP_FLAGS_SUPPRESSION_PARAM1
- ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM0
- ZI_IMP_FLAGS_STRONGCOMPENSATION_PARAM1
- 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

enum ZIVectorElementType_enum

Enumerates all the types that a ::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_ASCIIZ
  NULL-terminated string.

enum ZIAPIVersion_enum

Enumerator:

- ZI_API_VERSION_0
- ZI_API_VERSION_1
- ZI_API_VERSION_4
- ZI_API_VERSION_5

enum ZIListNodes_enum

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_NONE

  Default, return a simple listing of the given node immediate descendants.

- ZI_LIST_RECURSIVE

  List the nodes recursively.

- ZI_LIST_ABSOLUTE

  Return absolute paths.

- ZI_LIST_LEAFSONLY

  Return only leaf nodes, which means the nodes at the outermost level of the tree.

- ZI_LIST_SETTINGSONLY

  Return only nodes which are marked as setting.

enum ZIModuleHeaderType_enum

Enumerates all module header types.

Enumerator:

- ZI_MODULE_HEADER_TYPE_NONE
- ZI_MODULE_HEADER_TYPE_SWTRIGGER
- ZI_MODULE_HEADER_TYPE_SWEEPER

enum ZIVectorWriteStatus_enum

Enumerator:

- ZI_VECTOR_WRITE_STATUS_IDLE
- ZI_VECTOR_WRITE_STATUS_PENDING

enum TREE_ACTION

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

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.

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 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

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

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 [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIDoubleData ValueD;

  if ((RetVal = ziAPISetValueI(Conn,
                               "/dev1046/demods/*/rate",
                               100)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
  }

  if ((RetVal = ziAPIGetValueD(Conn,
                               "/dev1046/demods/0/rate",
```

```
                                      &ValueD)) != ZI_INFO_SUCCESS) {
      ziAPIGetError(RetVal, &ErrBuffer, NULL);
      fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
      printf("Value = %f\n", ValueD);
    }
  }
```

See Also:

ziAPISetValueD, 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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIIntegerData ValueI;

  if ((RetVal = ziAPISetValueD(Conn,
                               "/dev1046/demods/*/rate",
                               5.53)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
  }

  if ((RetVal = ziAPIGetValueI(Conn,
                               "/dev1046/demods/0/rate",
```

```
                                    &ValueI)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
  } else {
    printf("Value = %f\n", (float)ValueI);
  }
}
```

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 GetSample(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 GetSample(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 GetSample(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 doesn't reach 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

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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIIntegerData ValueI;

  if ((RetVal = ziAPISetValueD(Conn,
                               "/dev1046/demods/*/rate",
                               5.53)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
```

```
    }

    if ((RetVal = ziAPIGetValueI(Conn,
                                 "/dev1046/demods/0/rate",
                                 &ValueI)) != ZI_INFO_SUCCESS) {
      ziAPIGetError(RetVal, &ErrBuffer, NULL);
      fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
      printf("Value = %f\n", (float)ValueI);
    }
  }
```

See Also:

ziAPIGetValueD. ziAPISyncSetValueD

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.

```
// Copyright [2016] Zurich Instruments AG
#include <stdlib.h>
#include <stdio.h>

#include "ziAPI.h"

void UpdateValue(ZIConnection Conn) {
  ZIResult_enum RetVal;
  char* ErrBuffer;
  ZIDoubleData ValueD;

  if ((RetVal = ziAPISetValueI(Conn,
                               "/dev1046/demods/*/rate",
                               100)) != ZI_INFO_SUCCESS) {
    ziAPIGetError(RetVal, &ErrBuffer, NULL);
    fprintf(stderr, "Error, can't set Parameter: %s.\n", ErrBuffer);
```

```
    }

    if ((RetVal = ziAPIGetValueD(Conn,
                                 "/dev1046/demods/0/rate",
                                 &ValueD)) != ZI_INFO_SUCCESS) {
      ziAPIGetError(RetVal, &ErrBuffer, NULL);
      fprintf(stderr, "Error, can't get Parameter: %s.\n", ErrBuffer);
    } else {
      printf("Value = %f\n", ValueD);
    }
  }
```

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 ore 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

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 ore 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

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)

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.

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

ziAPIModSetIntegerData

ZIResult_enum ziAPIModSetIntegerData (ZIConnection conn, ZIModuleHandle handle, const char* path, ZIIntegerData 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

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

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 specifyin 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 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 callin 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 ppreallocates 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

ziAPIDiscoveryFind

ZIResult_enum ziAPIDiscoveryFind (ZIConnection conn, const char* deviceAddress, const char** deviceId)

Returns the device id for a given device address. Attention! Invalidates all pointers previously returned by ziAPIDiscovery* calls.

ziAPIDiscoveryGet

ZIResult_enum ziAPIDiscoveryGet (ZIConnection conn, const char* deviceId, const char** propsJSON)

Returns the device properties for a given device id in JSON format.

ziAPIDiscoveryGetValueI

ZIResult_enum ziAPIDiscoveryGetValueI (ZIConnection conn, const char* deviceId, const char* propName, ZIIntegerData* value)

Returns given integer property value for a given device id.

ziAPIDiscoveryGetValueS

ZIResult_enum ziAPIDiscoveryGetValueS (ZIConnection conn, const char* deviceId, const char* propName, const char value)**

Returns given string property value for a given device id.

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 $f_{cut-off}$ or 3dB frequency f_{-3dB}. 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(differential gain / 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.
GPIB	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: $U_{RMS} = U_{PK} / \sqrt{2} = U_{PK} / 1.41$. 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

A

C

D

L

M

N

P

S

U

Z