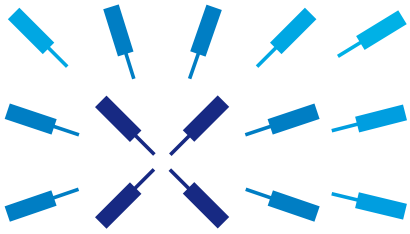
LabOne Programming Manual



Zurich Instruments

LabOne Programming Manual

Zurich Instruments AG

Revision 22.08

Table of Contents

What's New in the LabOne Programming Manual IV
1. Introduction 7
    1.1. LabOne Programming Quick Start Guide 8
    1.2. LabOne Software Architecture 11
    1.3. Comparison of the LabOne APIs 15
    1.4. Initializing a Connection to a Data Server 16
    1.5. Compatibility 21
2. Instrument Communication 22
    2.1. Data Server's Node Tree 23
    2.2. Data Streaming 31
    2.3. Comparison of Data Acquisition Methods 35
    2.4. Demodulator Sample Data Structure 36
    2.5. Instrument-Specific Considerations 37
3. LabOne API Programming 38
    3.1. An Introduction to LabOne Modules 39
    3.2. Low-level LabOne API Commands 41
    3.3. AWG Module 46
    3.4. Data Acquisition Module 58
    3.5. Device Settings Module 80
    3.6. Impedance Module 83
    3.7. Multi-Device Synchronisation Module 91
    3.8. PID Advisor Module 94
    3.9. Precompensation Advisor Module 108
    3.10. Quantum Analyzer Module 119
    3.11. Scope Module 123
    3.12. Sweeper Module 134
4. MATLAB Programming 148
    4.1. Installing the LabOne MATLAB API 149
    4.2. Getting Started with the LabOne MATLAB API 152
    4.3. LabOne MATLAB API Tips and Tricks 155
    4.4. Troubleshooting the LabOne MATLAB API 157
5. Python Programming 159
    5.1. Installing the LabOne Python API 160
    5.2. Getting Started with the LabOne Python API 162
    5.3. LabOne Python API Tips and Tricks 164
6. LabVIEW Programming 165
    6.1. Installing the LabOne LabVIEW API 166
    6.2. Getting Started with the LabOne LabVIEW API 168
    6.3. LabVIEW Programming Tips and Tricks 173
7. .NET Programming 174
    7.1. Installing the LabOne .NET API 175
    7.2. Getting Started with the LabOne .NET API 176
    7.3. LabOne .NET API Examples 180
8. C Programming 181
    8.1. Getting Started 182
    8.2. Error Handling and Logging in the LabOne C API 184
Glossary 185
Index 191

What's New in the LabOne Programming Manual

Release 22.08

Release date: 31-Aug-2022

Update of the LabOne Programming Manual for LabOne Release 22.08.

Highlights:

- SHFQC: Official support for the SHFQC Qubit Controller.
- Quantum Analyzer Module: Enabled record functionality.
- Python API: Renamed `zhinst.ziPython` to `zhinst.core`.
- Python API: Added custom exception types to `zhinst.core`.
- Python API: Added `compile_seqc()` to `zhinst.core` to compile AWG sequencer codes.
- Python API: Dropped support for Python 3.6.
- C API: Added `CMakeLists.txt` to conveniently build C API examples.
- MATLAB API: Enabled complex-valued vectors in Quantum Analyzer Module.
- LabVIEW API: Introduced two DAQ Module tutorials for single-shot and continuous data acquisition.

Release 22.02

Release date: 28-Feb-2022

Update of the LabOne Programming Manual for LabOne Release 22.02.

Highlights:

- LabOne UI & API: Include 'Flat Top' window function in FFT mode of Scope, DAQ, and Spectrum.
- LabOne Data Server: Node paths are returned in lower case by `listNodes` and `listNodesJSON` methods.
- Python API: Support of Python 3.10.
- MATLAB API: Examples are available in the public GitHub repository.
- C API: Transactional commands are enabled for multi-parameter setting.
- C API: Example for impedance data acquisition using poll and averaging.
- C API: Example for short-open user compensation using MFIA Impedance Analyzer.
- C API: Example to demonstrate acquisition of a single fresh value from a streaming node.

Release 21.08

Release date: 31-Aug-2021

Update of the LabOne Programming Manual for LabOne Release 21.08.

Highlights:

- Data Server, Webserver, and C API: Support for GNU/Linux and macOS on ARM64 architecture.
- LabOne API: Data Server log messages propagated to Client via API.
- Python API: Support of Python 3.7+ for Gnu/Linux on ARM64 processors.
- Python API: Support of Python 3.9 for macOS on Apple M1 processor.
- Python API: Examples moved from zhinst package to a public GitHub repository.

Release 21.02

Release date: 28-Feb-2021

Update of the LabOne Programming Manual for LabOne Release 21.02.

Highlights:

- Released Online Programming Manual and API Documentation.
- Data Server: Allow running multiple instances of data server on Windows.
- Python API: Added support of Python 3.9 and removed support of Python 2.7.
- Python API: Added API functions for adjusting debug level and log path for file output.
- MATLAB API: Added one example to demonstrate command table feature of HDAWG.

Release 20.07

Release date: 28-Aug-2020

Update of the LabOne Programming Manual for LabOne Release 20.07.

Highlights:

- All APIs: Improved error handling when setting an invalid wildcard path.
- Python and MATLAB APIs: Enumerated integer nodes can be set using keyword strings using the setString function to improve code readability.
- C API: Scope module support added.

Release 20.01

Release date: 28-Feb-2020

Update of the LabOne Programming Manual for LabOne Release 20.01.

Highlights:

- LabOne API: Added quantum analyzer API module
- LabOne API: improved error handling when a device is not connected or misspelled
- LabOne API: included full revision information as a packed decimal in the revision node
- API Log: removed module prefix from API log entries as it is no longer required
- Python API: added deprecation warning for users of Python versions 2.7 and 3.5
- C and .NET API: impedance flags added to 'ZISweeperImpedanceWave' class

Chapter 1. Introduction

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

- Section 1.1 for help LabOne programming quick start guide.
- Section 1.2 for an overview of the LabOne software architecture.
- Section 1.3 for a comparison of the LabOne APIs.
- Section 1.4 for help initializing a connection to a data server.

Instrument Specifics

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

HF2 Real-time Option

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

1.1. LabOne Programming Quick Start Guide

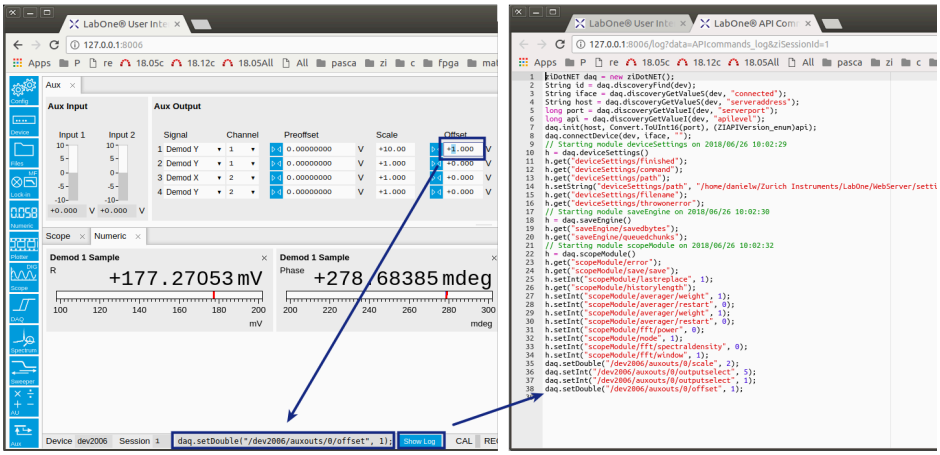
This section contains a collection of tips to help you get started programming with your instrument as quickly as possible.

Use the LabOne User Interface to develop measurement methods

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

Let the LabOne User Interface write code for you

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



Use Device Discovery to connect to your device

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

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

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

```
api_session = zhinst.core.ziDAQServer(props['serveraddress'], props['serverport'],
 props['apilevel'])
api_session.connectDevice(dev, props['interfaces'][0])
api_session.getDouble('/{}/demods/0/rate'.format(dev))
```

```
# Out: 1.81e3
```

Further explanation about creating an API session with the appropriate the Data Server is provided in Section 1.4.1.

Use the distributed examples as a base for your program

Each LabOne API contains many examples to help you get started. Our public GitHub repository (https://github.com/zhinst/labone-api-examples) provides a constantly updated collection of examples for the different APIs.

The following APIs provide built-in examples: MATLAB, LabVIEW, .NET, C.

Use the API's logging capabilities

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

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

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

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

These commands generate the log:

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

See the API-specific chapter for more details:

- Enabling Logging in the LabOne MATLAB API
- Enabling Logging in the LabOne Python API
- Error Handling and Logging in the LabOne C API

Use the API utility functions

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

Load LabOne User Interface settings files from the APIs

The XML files used for device settings can not only be loaded and saved from the LabOne User Interface but from any of the APIs. See Section 3.5 for more information.

1.2. LabOne Software Architecture

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

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

This software architecture is organized in layers, see Figure 1.1 for a schematic of the software layers.

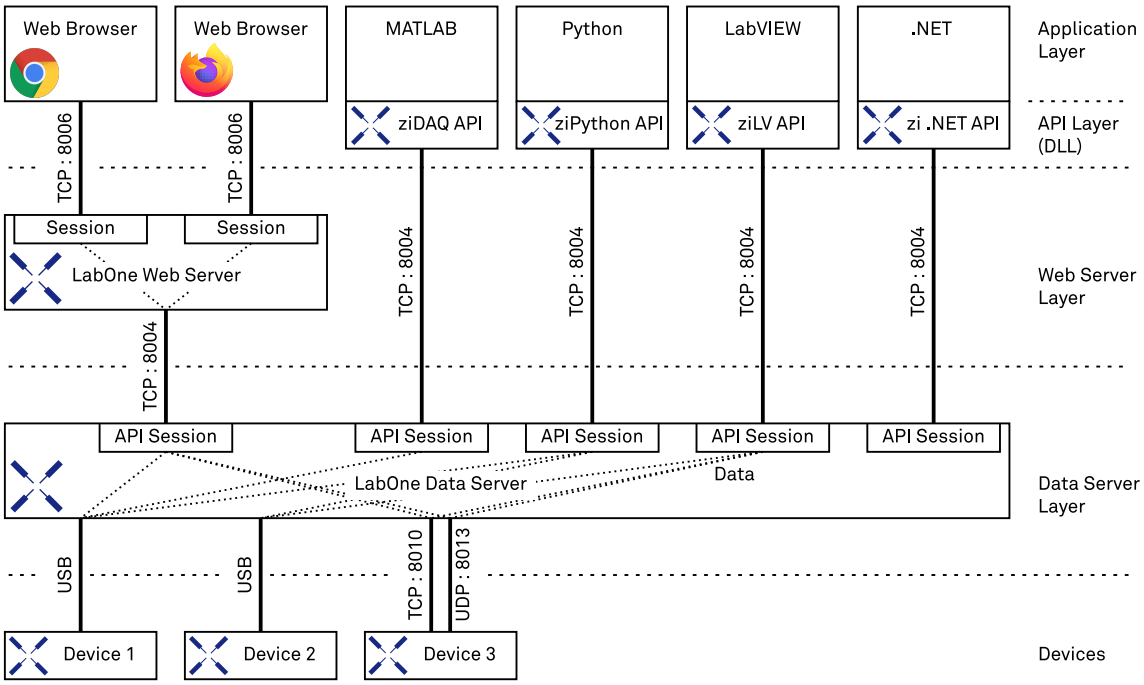


Figure 1.1. LabOne Software Architecture. The above diagram depicts the software architecture when using Zurich Instruments. In the case of MFLI and MFIA instruments the Data Server and/or the Web Server can also run on the device itself instead of on a host computer (see MFLI / MFIA Software Configuration).

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

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

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

1.2.1. MFLI /MFIA Software Configuration

In their simplest form, The MFLI/MFIA instruments are self contained. The LabOne Web Server and Data Server run on the instrument itself. Only the LabOne APIs run on an external PC.

However, to improve performance, other software configurations are possible. By installing the LabOne software on an external PC, the MFLI/MFIA instrument can be accessed via a LabOne Web Server running there. This gives advantages due to the improved computing power and the greater memory resources of the external PC. Moreover, if performing measurements with two or more synchronized MFLI/MFIA instruments, a LabOne Data Server running on the external PC can also be used. Indeed, to synchronize multiple MFLI/MFIA instruments, this software configuration is mandatory.

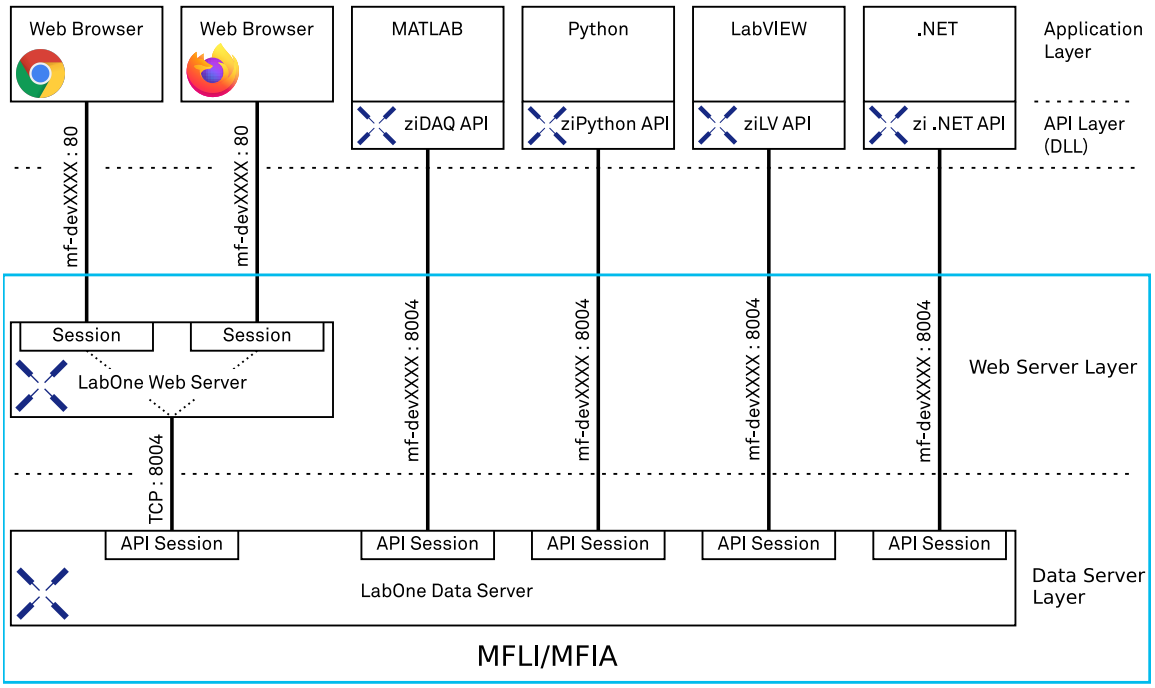


Figure 1.2. LabOne Software Configuration MFLI/MFIA. The above diagram shows the simplest software configuration for the MFLI/MFIA instruments. The Data Server and Web Server run on the device itself (devXXXX represents the serial number of the instrument).

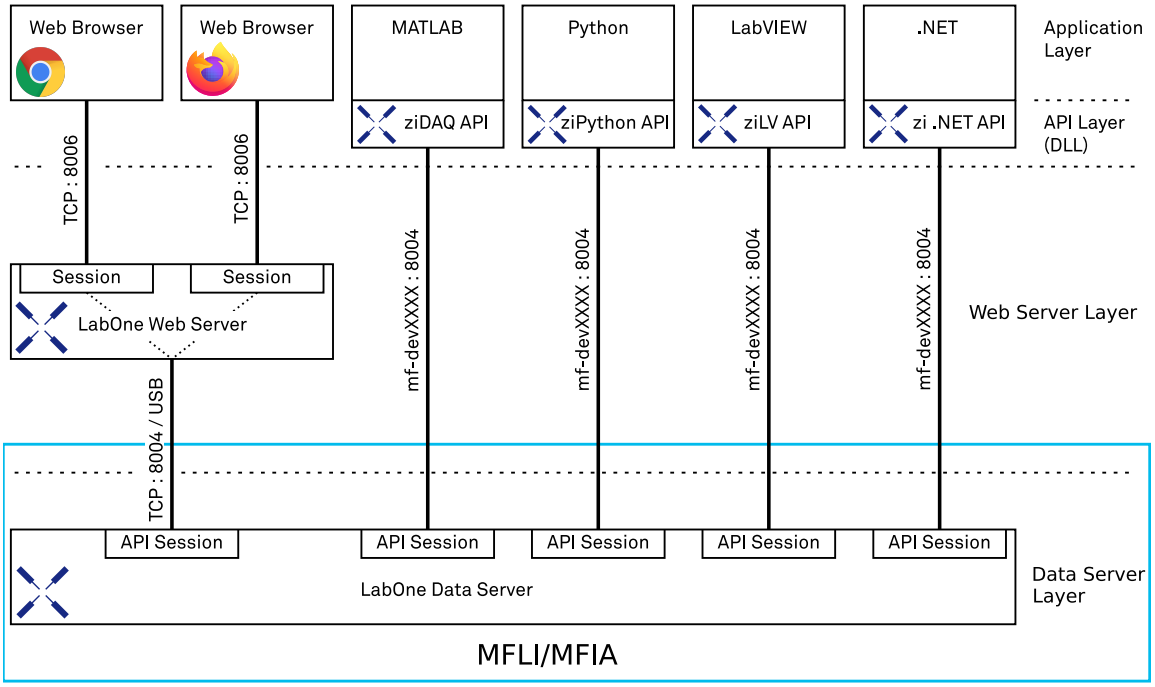


Figure 1.3. LabOne Software Configuration for the MFLI with the LabOne Web Server running on an external PC (devXXXX represents the serial number of the instrument).

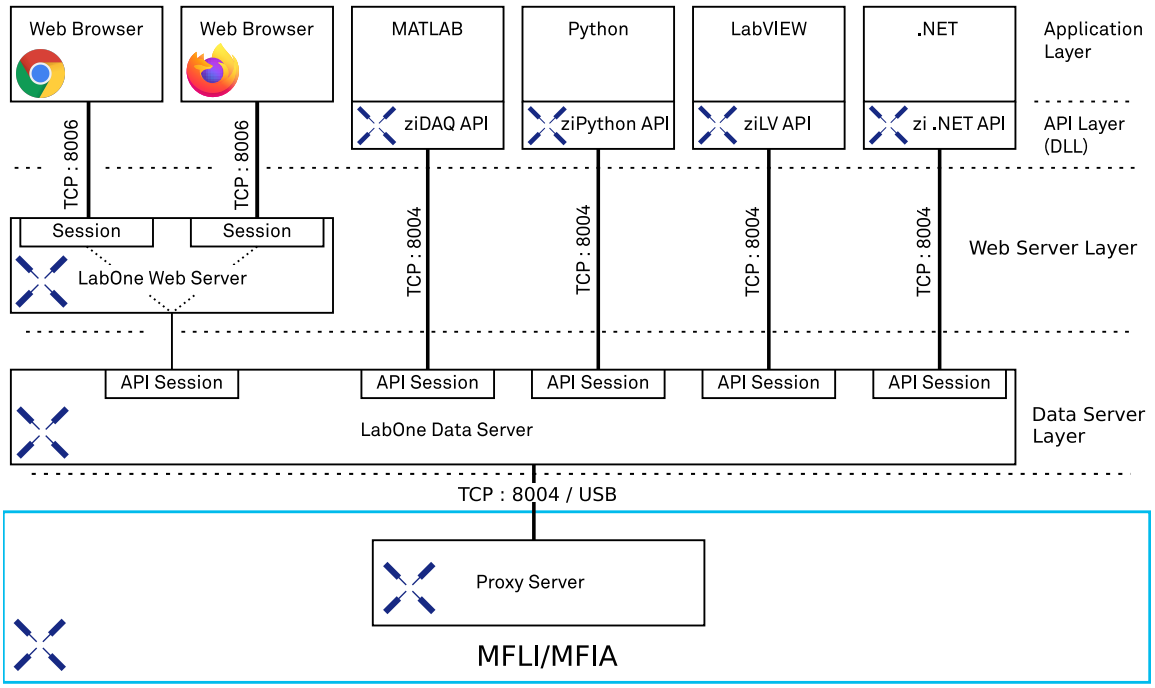


Figure 1.4. LabOne Software Configuration for the MFLI with LabOne Web Server and Data Server running on an external PC. This software configuration is mandatory when synchronizing multiple MFLI/MFIA instruments.

1.3. Comparison of the LabOne APIs

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

- The LabVIEW Programming 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 Programming 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 Programming allows the user to directly obtain measurement data within python. Python is available as free and open source software; no license is required to use it.
- The .NET Programming allows the user to directly obtain measurement data within the .NET programming framework using the C#, Visual Basic or #F programming languages. To use the .NET API a Microsoft Visual Studio installation is required.
- The C API is a very versatile interface that will run on most platforms. However, since C is a low-level programming language, the development cycle is slower than with the other programming environments.
- The text-based interface (HF2 Series only) allows the user to manually connect to the HF2 Data Server in a console via telnet. While this interface is a very useful tool for HF2 programmers to verify instrument configuration set by other interfaces, it is limited in terms of performance and maximum demodulator sample rate. See the HF2 User Manual for more details.

Note

From LabOne Release 15.05 onwards the Sweeper and DAQ (formerly called Software Trigger) Modules are also available in the LabVIEW and C APIs and from 16.12 onwards all Modules are available. All modules were previously available in the MATLAB and Python LabOne APIs.

1.4. Initializing a Connection to a Data Server

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

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

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

1.4.1. Specifying the Data Server Hostname and Port

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

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

Data Server Port

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

Data Server Hostname (HDAWG, HF2 and UHF Instruments)

In the simplest configuration for HDAWG, HF2 and UHF instruments, the instrument is attached to the same PC where both the data server and API client are running. Since the API client is running on the same PC as the data server, the `'localhost'` (equivalently, `'127.0.0.1'`) should be specified as the data server address, Figure 1.5.

The API client may also connect to a data server running on a different PC from the client. In this case, the data server address should be the IP address (or hostname, if available) of the PC where the data server is running. Note, remote data server access is not enabled by default and the data server must be configured in order to listen to non-localhost connections by either enabling the `--open-override` command-line option when starting the data server or by setting the value of the server node `/zi/config/open` to 1 on a running data server (clearly only possible from a client running on the localhost). See Section 2.1.2 for more information on nodes.

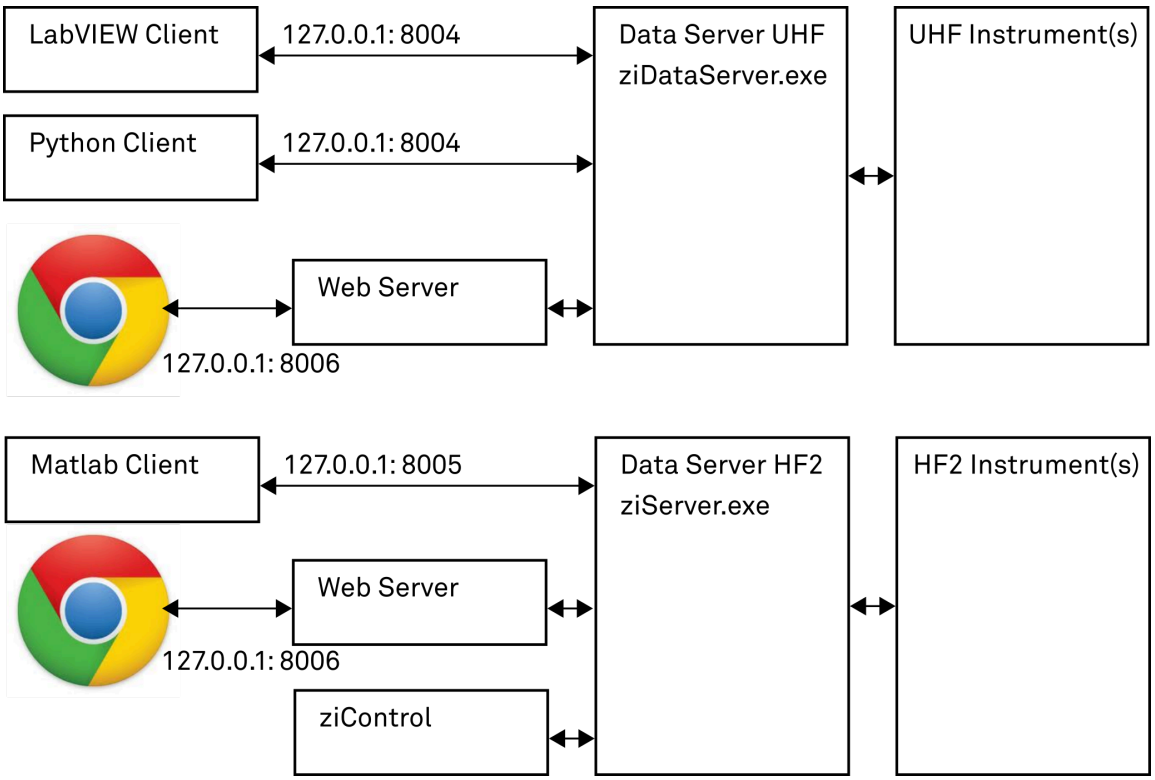


Figure 1.5. Server address and port handling for HDAWG, HF2 and UHF instruments for the case where the API client and data server are running on the same PC. In this case the server hostname is `localhost` and the default port value is 8004 for HDAWG and UHF Instruments and 8005 for HF2 Instruments.

Data Server Hostname (MF Instruments)

In the case of MF instruments the data server runs on the instrument itself and as such an API client from a PC always accesses the data server remotely. Thus, in this case the data server hostname is that of the instrument itself. 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 1.6.

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

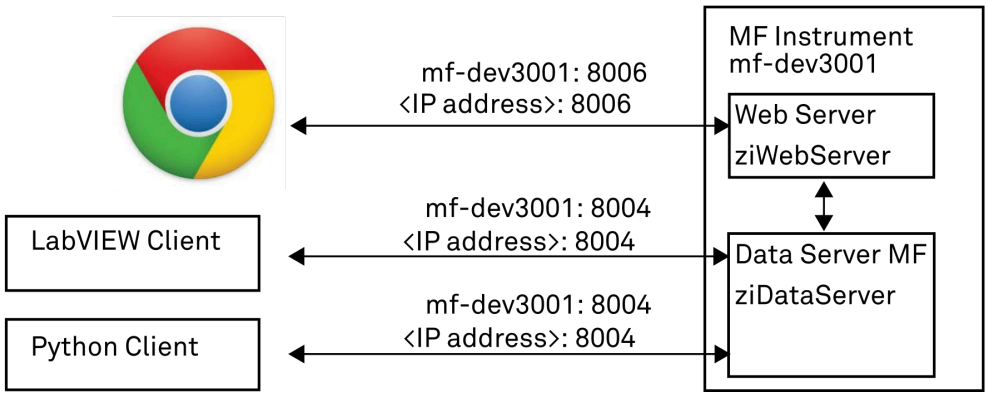


Figure 1.6. Server address and port handling on MF Instruments. The data server runs on the instrument and the server hostname is the same as the instrument's hostname. Using a hostname of the form mf-dev3001 is only applicable when using the 1GbE interface. The default data server port is 8004 for MF Instruments.

API Level and Connectivity Examples

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

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

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

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

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

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

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

Working in a Multi-threaded Program

It is important to note that API session objects are not thread-safe, i.e. a single API session cannot be shared between multiple client threads. If you want to use a LabOne API in a multi-threaded program, for each client thread, use a separate API session.

1.4.2. LabOne API Levels

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

software release. Occasionally, however, we do have to make a breaking change in our API by removing some old functionality. This old functionality is, however, phased out over several software releases. First, the functionality is marked as deprecated and the user is informed via a deprecation 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 only uses 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 to the existing code.

The current available API levels are:

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

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

Note

The HF2 Series only supports API Level 1.

Note

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

API Level 4 Features

The new features in API Level 4 are:

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

API Level 5 Features

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

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

API Level 6 Features

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

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

1.5. Compatibility

Controlling an instrument requires the combination of several software components: The instrument's firmware, a Data Server and an API. In general, whenever possible, it is recommended to use the latest (and same) software release version (e.g., "20.01") of all these components. If you are bound to a certain version for technical reasons, then it is recommended to use the same version of all components. However, this is not strictly necessary in all cases. If it is absolutely necessary to mix versions, this section explains how to verify whether different versions of various software components may be mixed with each other.

1.5.1. API and Data Server Compatibility

Although it is recommended to use the same software release version (e.g. "20.01") of both API and Data Server, it is not strictly necessary. The interface between API and Data Server remains the same between versions. However, there may be a change in some Section 2.1.2 that effects specific functionality.

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

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

Chapter 2. Instrument Communication

This section describes the main concepts in LabOne software that allow high-speed data acquisition and guides the user to the best acquisition method for their measurement task.

It is divided into sub-sections as follows:

- Section 2.1 explains how device settings and data are organized and accessible in Data Server's Node Tree.
- Section 2.2 explains Zurich Instruments' data streaming concept for data acquisition.
- Section 2.3 compares the methods available for comparison of data acquisition methods.
- Section 2.5 documents some Instrument-specific considerations.

2.1. Data Server's Node Tree

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

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

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

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

/DEV2006/AUXOUTS/

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

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

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

/DEV2006/AUXOUTS/0/DEMODSELECT
/DEV2006/AUXOUTS/0/LIMITLOWER
/DEV2006/AUXOUTS/0/LIMITUPPER
/DEV2006/AUXOUTS/0/OFFSET
/DEV2006/AUXOUTS/0/OUTPUTSELECT
/DEV2006/AUXOUTS/0/PREOFFSET
/DEV2006/AUXOUTS/0/SCALE
/DEV2006/AUXOUTS/0/VALUE

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

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

Module Parameters

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

2.1.1. Node Properties and Data Types

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

Read	Data can be read from the node.
Write	Data can be written to the node.
Setting	The node corresponds to a writable instrument configuration. The data of these nodes are persisted in snapshots of the instrument and stored in the LabOne XML settings files.
Streaming	A node with the read attribute that provides instrument data, typically at a user-configured rate. The data is usually a more complex data type, for example demodulator data is returned as `ZIDemodSample`. A full list of streaming nodes is available in Table 2.1. Their availability depends on the device class (e.g. MF) and the option set installed on the device.

A node may contain data of the following types:

Integer	Integer data.
Double	Double precision floating point data. Note that the actual value on the device may only be calculated in single precision.
String	A string array.
Enumerated (integer)	As for Integer, but the node only allows certain values.
Composite data type	For example, `ZIDemodSample`. These custom data types are structures whose fields contain the instrument output, a timestamp and other relevant instrument settings such as the demodulator oscillator frequency. Documentation of custom data types is available in Chapter 8. the C Programming Chapter in the LabOne Programming manual.

2.1.2. Exploring the Node Tree

In the LabOne User Interface

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

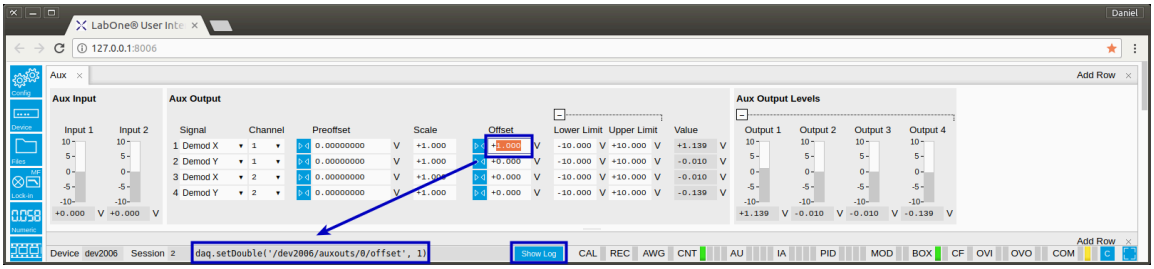


Figure 2.1. When a device's configuration is modified in the LabOne User Interface, the Status Bar displays the equivalent command to perform the same configuration via a LabOne programming interface. Here, the MATLAB code to modify auxiliary output 1's offset value is provided. When "Show Log" is clicked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

In the Instrument-specific User Manual

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

In a LabOne Programming Interface

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

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

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

2.1.3. Data Server Nodes

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

- /ZI/ABOUT/VERSION
- /ZI/ABOUT/REVISION

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

The nodes:

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

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

Nodes that are of particular use to programmers are:

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

2.1.4. Reference Node Documentation

This section describes all the nodes in the data server's node tree organized by branch.

ZI (LabOne Data Server Nodes)

/ZI/ABOUT/COMMIT

Properties:	Read
Type:	String
Unit:	None

Contains the commit hash of the source code used to build this version of the LabOne software.

/ZI/ABOUT/COPYRIGHT

Properties:	Read
Type:	String
Unit:	None

Holds the copyright notice.

/ZI/ABOUT/DATASERVER

Properties:	Read
Type:	String
Unit:	None

Contains information about the Zurich Instruments Data Server.

/ZI/ABOUT/FWREVISION

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

Contains the revision of the device firmware.

/ZI/ABOUT/REVISION

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

Contains the revision number of the Zurich Instruments Data Server.

/ZI/ABOUT/VERSION

Properties:	Read
Type:	String
Unit:	None

Contains the version of the LabOne software.

/ZI/CLOCKBASE

Properties:	Read
Type:	Double
Unit:	None

A fallback clock frequency that can be used by clients for calculating time bases when no other is available.

/ZI/CONFIG/OPEN

Properties:	Read, Write, Setting
Type:	Integer (enumerated)
Unit:	None

Enable communication with the LabOne Data Server from other computers in the network.

0	`local` Communication only possible with the local machine.
1	`network`

Communication possible with other machines in the network.

/ZI/CONFIG/PORT

Properties: Read
Type: Integer (64 bit)
Unit: None

The IP port on which the LabOne Data Server listens.

/ZI/DEBUG/LEVEL

Properties: Read, Write, Setting
Type: Integer (enumerated)
Unit: None

Set the logging level (amount of detail) of the LabOne Data Server.

0 `trace`
Trace. Messages designated as traces are logged.

1 `debug`
Debug. Messages designated as debugging info are logged.

2 `info`
Info. Messages designated as informational are logged.

3 `status`
Status. Messages designated as status info are logged.

4 `warning`
Warning. Messages designated as warnings are logged.

5 `error`
Error. Messages designated as errors are logged.

6 `fatal`
Fatal. Messages designated as fatal errors are logged.

/ZI/DEBUG/LOG

Properties: Read
Type: String
Unit: None

Returns the logfile text of the LabOne Data Server.

/ZI/DEBUG/LOGPATH

Properties: Read

Type: String
Unit: None

Returns the path of the log directory.

/ZI/DEVICES/CONNECTED

Properties: Read
Type: String
Unit: None

Contains a list of devices connected to the LabOne Data Server.

/ZI/DEVICES/DISCOVER

Properties: Read, Write
Type: String
Unit: None

Not used.

/ZI/DEVICES/VISIBLE

Properties: Read
Type: String
Unit: None

Contains a list of devices in the network visible to the LabOne Data Server.

/ZI/MDS/GROUPS/n/DEVICES

Properties: Read, Write, Setting
Type: String
Unit: None

Contains a list of devices in this synchronization group.

/ZI/MDS/GROUPS/n/KEEPALIVE

Properties: Read, Write, Setting
Type: Integer (64 bit)
Unit: None

Set by the MDS module to indicate control over this synchronization group.

/ZI/MDS/GROUPS/n/LOCKED

Properties:	Read, Write, Setting
Type:	Integer (64 bit)
Unit:	None

Indicates whether the device group is locked by a MDS module.

/ZI/MDS/GROUPS/n/STATUS

Properties:	Read, Write, Setting
Type:	Integer (enumerated)
Unit:	None

Indicates the status the synchronization group.

-1	Error. An error occurred in the synchronization process.
0	New
1	Sync
2	Alive

2.2. Data Streaming

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

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

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

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

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

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

The following sections explain these points in further detail, users who acquire the data from Data Acquisition Module may prefer to skip ahead Section 2.3.

2.2.1. Streaming Nodes

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

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

The qualifier "continuous" deserves further explanation. Each sample point is a discrete data point sent from the device. Continuous means that the samples are not only continually sent,

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

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

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

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

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

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

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

2.2.2. Alignment of Streaming Node Data

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

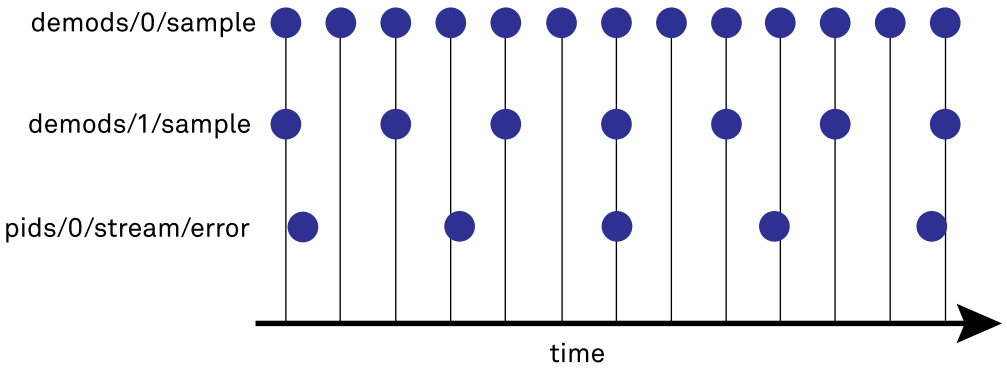


Figure 2.2. Samples from different streaming nodes configured with different rates: Demod 1 at 2N kSamples/s, Demod 2 at N kSamples/s and PID Error 1 at M kSample/s (N not divisible by M). Although each stream consists of equidistantly samples in time, the sample timestamps from different streams are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

2.2.3. Data Loss

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

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

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

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

2.3. Comparison of Data Acquisition Methods

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

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

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

Scope Data

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

2.4. Demodulator Sample Data Structure

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

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 8 contains details of other data structures.

2.5. Instrument-Specific Considerations

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

2.5.1. MF-Specific Considerations

Identifying which Data Server the MF device is connected to

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

2.5.2. UHF-Specific Considerations

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

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

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

Chapter 3. LabOne API Programming

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

- Section 3.1 for an Introduction to LabOne Modules.
- Section 3.2 for Low-level LabOne API Commands.
- Section 3.3 for the AWG Module.
- Section 3.4 for the Data Acquisition Module.
- Section 3.5 for the Device Settings Module.
- Section 3.6 for the Impedance Module.
- Section 3.7 for the Multi-Device Synchronisation Module.
- Section 3.8 for the PID Advisor Module.
- Section 3.9 for the Precompensation Advisor Module.
- Section 3.10 for the Quantum Analyzer Module.
- Section 3.11 for the Scope Module.
- Section 3.12 for the Sweeper Module.

3.1. An Introduction to LabOne Modules

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

3.1.1. Software Architecture

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

3.1.2. ziCore Modules

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

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

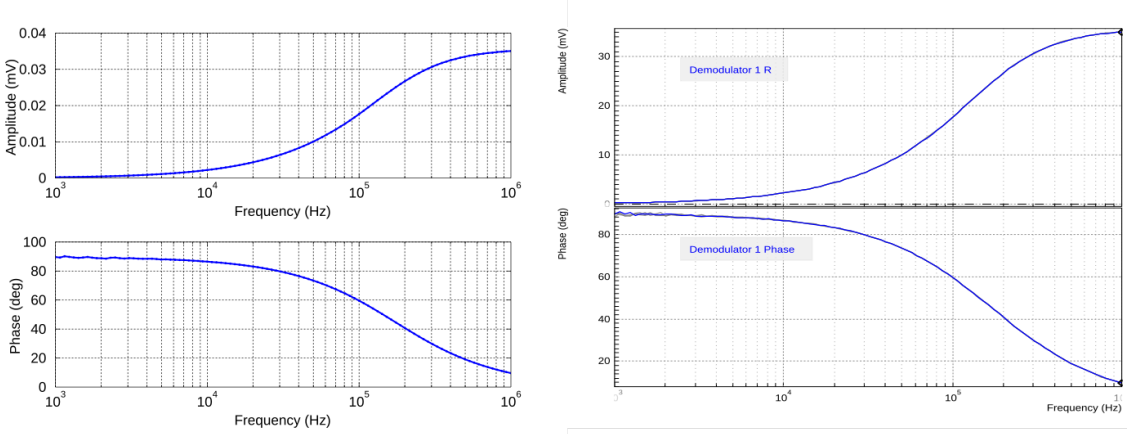


Figure 3.1. The same results and behavior can be obtained from Modules in any ziCore-based interface; Sweeper Module results from the LabOne MATLAB API (left) and the LabOne User Interface (right) using the same Sweeper and instrument settings.

The modules currently available in **ziCore** are:

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

- The Device Settings Module for saving and loading instrument settings to and from (XML) files.
- The PID Advisor Module for modeling and simulating the PID incorporated in the instrument.
- The Scope Module for obtaining scope data from the instrument.
- The Impedance Module for performing impedance measurements.
- The Multi-Device Synchronisation Module for synchronizing the timestamps of multiple instruments.
- The AWG Module for working with the AWG.
- The Precompensation Advisor Module also for working with the AWG.

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

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

3.1.3. Synchronous versus Asynchronous Commands

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

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

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

3.2. Low-level LabOne API Commands

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

3.2.1. The getSample command: For one-shot measurement demodulator data

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

Please note, the `getSample` function only works with the demodulator data type. It does not work with other data types such as impedance or auxiliary input samples. For non-demodulator sample types the recommended way to get data is via the subscribe and poll commands.

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

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

3.2.2. The subscribe and poll commands: For high-performance continuous or block streaming data

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

Simple Example

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

```
import zhinst.core

daq = zhinst.core.ziDAQServer('localhost', 8004, 6)
# Enable the demodulator output and set the transfer rate.
# This ensure the device actually pushes data to the Data Server.
daq.setInt('/dev2006/demods/0/enable', 1)
daq.setInt('/dev2006/demods/4/enable', 1)
```

```
# This value will be corrected to the nearest legal value by the instrument's FW.
daq.setDouble('/dev2006/demods/0/rate', 10e3)
daq.setDouble('/dev2006/demods/4/rate', 10e3)
daq.subscribe('/dev2006/demods/0/sample')
daq.subscribe('/dev2006/demods/4/sample')
time.sleep(1)  # Subscribed data is being accumulated by the Data Server.
data = daq.poll(0.020, 10, 0, True)
data.keys()
# Out: dict_keys(['/dev2006/demods/0/sample', '/dev2006/demods/4/sample'])
len(data['/dev2006/demods/0/sample']['timestamp'])
# Out: 13824
len(data['/dev2006/demods/4/sample']['timestamp'])
# Out: 13746
```

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

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

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

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

Ensure synchronization of settings before streaming data (sync)

To ensure that any settings have taken effect on the instrument before streaming data is processed, a special sync command is provided which ensures that the API blocks during the full

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

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

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

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

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 the timeout time passed, `poll` will either simply return an empty data structure (for example, an empty dictionary in Python) or throw an error, depending which flags it have been provided.

Note

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

Explanation of buffering in the Data Server and API Client

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

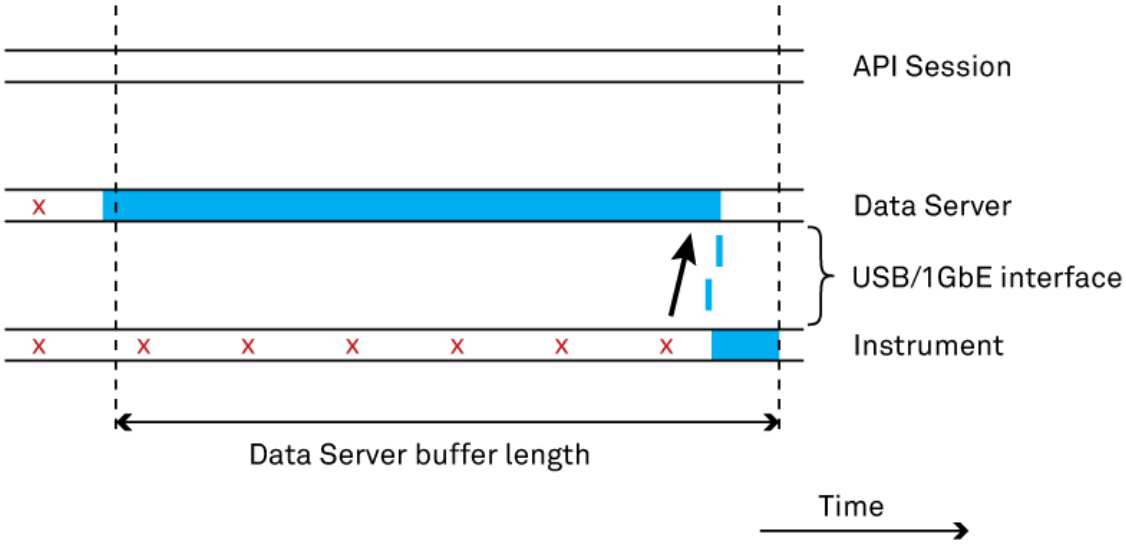


Figure 3.2. Illustration of data storage and transfer: the API Session (no other Data Server clients) is subscribed to a node (blue bars representing data stream) but never issues a `poll` command. The data are stored in the Data Server's buffer for a certain time and dumped afterwards.

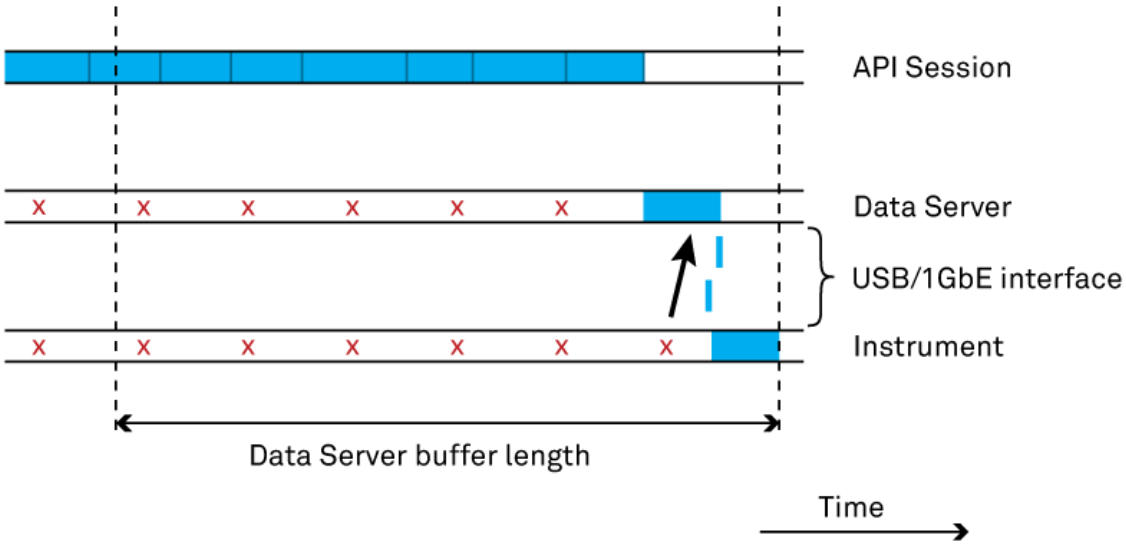


Figure 3.3. Illustration of data storage and transfer: the API Session is subscribed to a node and regularly issues a `poll` command. The Data Server holds only the data in the memory that were accumulated since the last `poll` command.

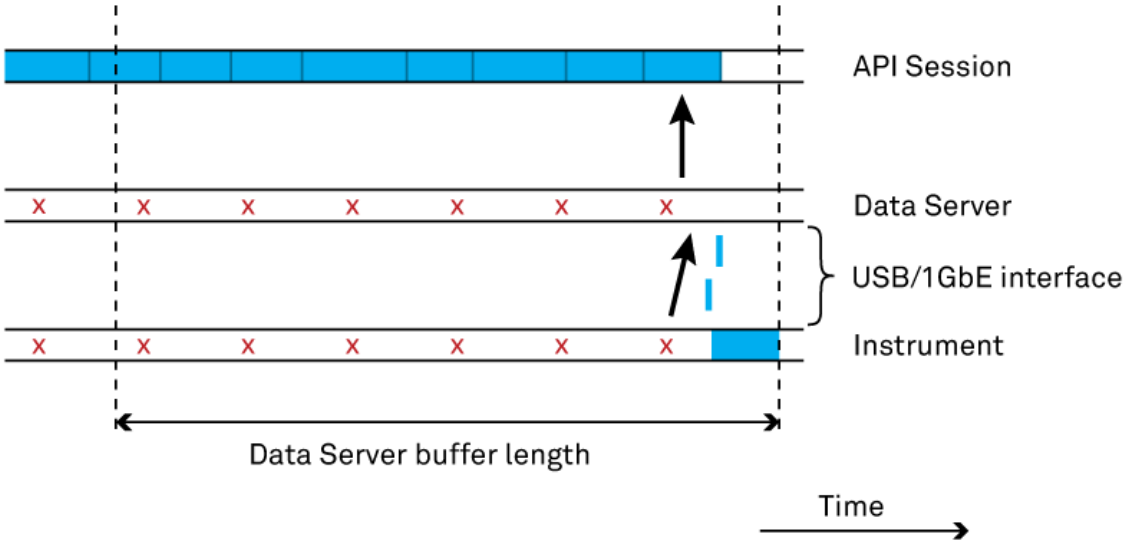


Figure 3.4. Illustration of data storage and transfer: the API Session is subscribed to a node and regularly issues a `poll` command. Upon a new `poll` command, all data accumulated in the Data Server buffer are transferred to the API Session and subsequently cleared from the Data Server buffer.

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

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

3.3. AWG Module

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

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

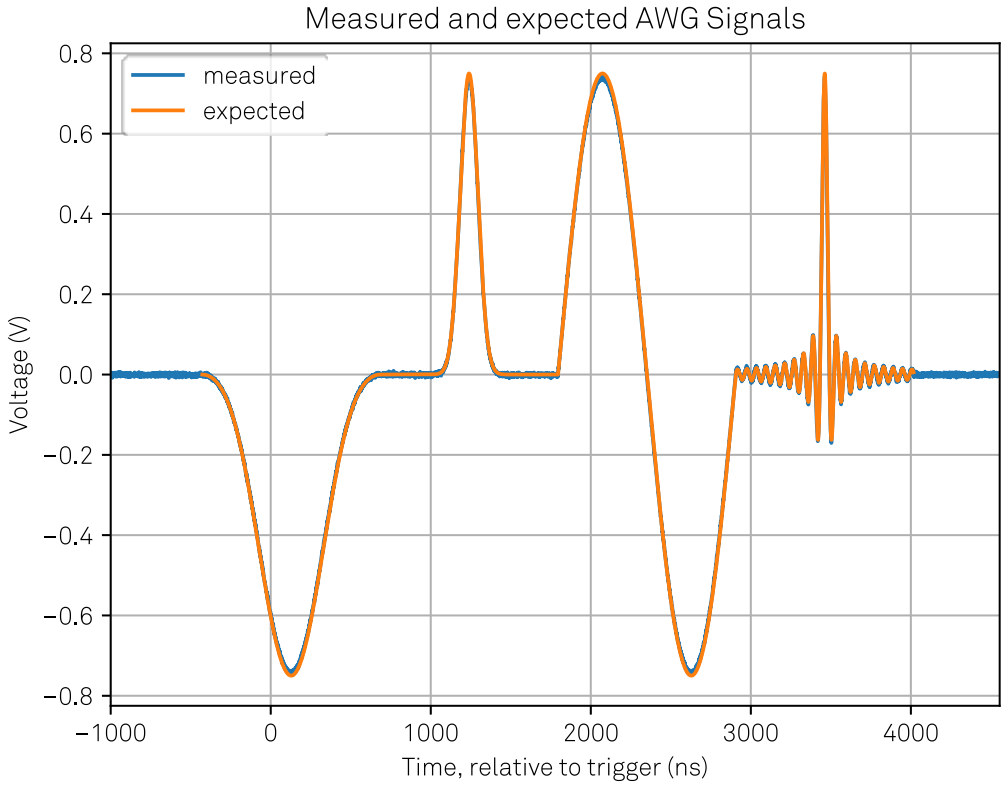


Figure 3.5. An AWG signal generated and measured using a UHFLI with the AWG Option. The waveform data was measured via a feedback cable using the UHF's Scope. The plot was generated by the Python API example for the UHFLI, `example_awg.py`, which also generates the expected waveform and cross-correlates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

3.3.1. Getting Started with the AWG Module

The following API examples demonstrating AWG Module use are available:

- MATLAB and Python, **`example_awg.{py,m}`**: Compiles and uploads an AWG sequencer program from a string. It demonstrates how to define waveforms using the four methods listed below in Section 3.3.3. Separate versions of these examples are available for both UHF and HDAWG instruments.
- MATLAB and Python, **`example_awg_sourcefile.{py,m}`**: Demonstrates how to compile and upload an AWG sequencer program from a **`.seqc`** file. Separate versions of these examples are available for both UHF and HDAWG instruments.

- LabVIEW, `ziExample-UHFLI-Module-AWG.vi`: Compiles and uploads an AWG sequencer program from a string and captures the generated waveform in the scope (UHF only).
- .NET, `ExampleAwgModule()` (in `Examples.cs`): Compiles and uploads an AWG sequencer program from a string. It demonstrates how to define waveforms using the four methods listed below in Section 3.3.3.
- C API, `ExampleAWGUpload.c`: Demonstrates how to compile and upload an AWG sequencer program from a `.seqc` file.

3.3.2. Sequencer Program Compilation and Upload

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

Compilation

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

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

Upload

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

3.3.3. Methods to define Waveforms in Sequencer Programs

The waveforms played by an AWG sequencer program can be defined, or in the last case, modified, using the following four methods. These methods are demonstrated by the examples listed in Section 3.3.1.

1. By using one of the waveform generation functions such as `sine()`, `sinc()`, `gauss()`, etc. defined in the LabOne AWG Sequencer programming language. See the UHF, HDAWG or SHFSG User Manual for full reference documentation.
2. By defining a `placeholder` waveform and later loading the actual waveform from the API. This method is optimal, especially for long waveforms.[1]

[1]The UHF platform is an exception and this method is on the contrary slower. However, since the waveform memory is limited, that should not be relevant for most of the practical purposes.

3. By defining a waveform in a file, either with a floating point format in a CSV file, or a binary file.

API vector transfer

The waveform must be first defined in the sequence, either as `placeholder` or as completely defined waveform with valid samples. In the first case, the compiler will only allocate the required memory and the waveform content is loaded later. The waveform definition must specify the length and eventual presence of markers. This should be respected later when the actual waveform is loaded.

The waveform can be loaded from the API using the `set` command to write waveform data the following nodes:

UHF	/DEV.../AWGS/0/WAVEFORM/WAVES/<index>
HDAWG	/DEV.../AWGS/[0-3]/WAVEFORM/WAVES/<index>
SHFSG	/DEV.../SGCHANNELS/[0-7]/AWG/WAVEFORM/WAVES/<index> [a]

[a]Only dual-channel waveforms

These nodes are the same regardless of the channel grouping mode on the HDAWG, so for example even when using the 1x8 mode on HDAWG8, waveforms are addressed separately for all 4 AWG cores. Multiple waveform uploads can be combined into one `set` command, which reduces upload time (see below). The assignment of a waveform to an index is done directly in the AWG sequence program using the `assignWaveIndex` instruction. As example

```
wave w = placeholder(WFM_SIZE);
assignWaveIndex(1,w, INDEX);
```

will allocate a waveform of `WFM_SIZE` at the index `INDEX`.

Dual-channel waveforms

Dual-channel waveform are effectively a single waveform, so there is only one index associated:

```
wave w1 = placeholder(WFM_SIZE);
wave w2 = placeholder(WFM_SIZE);
assignWaveIndex(1,w1, 2,w2, INDEX);
```

HDAWG grouped mode

When the HDAWG is configured to work in grouped mode (all modes except 4x2 mode), see section Section 3.3.4, the waveforms for each AWG core are treated separately, even when they are played together. As in the 4x2 mode, waveforms can be either single- or dual-channel per core. As an example, this sequence

```
//AWG CORE 1
wave w1 = placeholder(WFM_SIZE);
wave w2 = placeholder(WFM_SIZE);
assignWaveIndex(1,w1, 2,w2, INDEX1);

//AWG CORE 2
wave w3 = placeholder(WFM_SIZE);
assignWaveIndex(3,w3, INDEX2);

//Playback
playWave(1,w1, 2,w2, 3,w3);
```

defines one dual-channel waveform on core 1, and one single-channel waveform on core 2. They can be loaded by writing their content to the nodes

- /DEV.../AWGS/0/WAVEFORM/WAVES/<INDEX1>
- /DEV.../AWGS/1/WAVEFORM/WAVES/<INDEX2>

The waveform nodes use the internal raw format of the instrument and map the hardware capabilities of an AWG core. Thus, each waveform node can hold up to two analog waveforms and four markers. The length, number of waves and the presence of markers must be the same as defined in the sequence. An analog waveform is represented as array of signed int16. The markers are represented as array of int16, with the marker values defined in the four LSB; the other 12 bits must be zeros (see Figure 3.6).

```
wave_int16 = int16((1 << 15 - 1) * wave_float);
markers = int16(mk1_out1 * 1 << 0 + mk2_out1 * 1 << 1 +
                mk1_out2 * 1 << 2 + mk2_out2 * 1 << 3);
```

If there is more than one analog waveform and/or markers, the arrays representing them must be interleaved; the order should be the first wave, then the second and finally the markers (see Figure 3.6).

Figure 3.6. Interleaving of waves and markers in AWG raw format

In Python, it's convenient to use the helper functions **zhinst.utils.convert_awg_waveform** and **zhinst.utils.parse_awg_waveform** to write and read these nodes.

When uploading 2 or more waveforms with the Python API, it is recommended to perform the waveform upload with a single **set** command. This is possible by combining multiple pairs of waveform addresses and data as a Python list of tuples, and using this list as the argument of the **set** command. In this way, the overhead in communication latency is paid only once, and waveform upload is much faster than when issuing a **set** command for each waveform. The example below shows both the usage of the helper functions, and the combination of multiple waveform uploads in one **set** command.

Note

The **set** command is only available with the Python API. Other APIs are limited to the use of **setVector**. **setVector** does not support combining multiple commands into one. Apart from that, its usage is identical to that of **set**.

```
import zhinst.core
import zhinst.utils
import numpy as np

device = 'dev8000'
daq = zhinst.core.ziDAQServer('localhost', 8004, 6) #Connect to the dataserver
daq.connectDevice(device, '1GbE')          #Connect to the device

#Generate a waveform and marker
WFM_SIZE = 1024
wave_a = np.sin(np.linspace(0, 10*np.pi, WFM_SIZE))*np.exp(np.linspace(0, -5,
 WFM_SIZE))
wave_b = np.sin(np.linspace(0, 20*np.pi, WFM_SIZE))*np.exp(np.linspace(0, -5,
 WFM_SIZE))
wave_c = np.sin(np.linspace(0, 30*np.pi, WFM_SIZE))*np.exp(np.linspace(0, -5,
 WFM_SIZE))
```

```
marker_a  = np.concatenate([  0b11*np.ones(32),
 np.zeros(WFM_SIZE-32)]).astype(int)
marker_bc = np.concatenate([0b1111*np.ones(32),
 np.zeros(WFM_SIZE-32)]).astype(int)
#Convert and send them to the instrument
wave_raw_a = zhinst.utils.convert_awg_waveform(wave_a, markers=marker_a)
wave_raw_bc = zhinst.utils.convert_awg_waveform(wave_b, wave_c, markers=marker_bc)

INDEX1 = 0
INDEX2 = 1

set_cmd = [(f'/{device:s}/awgs/0/waveform/waves/{INDEX1}', wave_raw_a),
           (f'/{device:s}/awgs/0/waveform/waves/{INDEX2}', wave_raw_bc) ]

daq.set(set_cmd)
```

The Python code above corresponds to an AWG sequence program as the one below, which makes use of a single-channel and a dual-channel waveform playback.

```
wave w_a = placeholder(WFM_SIZE, true, true);
wave w_b = placeholder(WFM_SIZE, true, true);
wave w_c = placeholder(WFM_SIZE, true, true);
assignWaveIndex(1, w_a,   INDEX1);
assignWaveIndex(w_b, w_c, INDEX2);

playWave(1, w_a);
playWave(w_b, w_c);
```

Note

The waveform nodes have the property SILENTWRITE. This means that subscribing to such a node has no effect, i.e. changes to the node will not be returned in `poll`. To obtain the contents of such nodes, `getAsEvent` has to be called followed by `poll`. For short vectors `get` may be used.

File on disk

A waveform stored on the disk can be loaded from the sequence by referring to the file name without extension in the sequence program. For example

```
//Definition inline with playWave
playWave("wave_file");

//Assign first to a wave data type, then use
wave w = "wave_file";
playWave(w)
```

Two formats are supported, an CSV ASCII based and a binary one. The binary format should be preferred as it offers faster compilation than the CSV format. The waveform files must be located in the "awg/waveforms" sub-directory of the LabOne user directory (see explanation of `directory`).

Binary

The binary format support only single-channel definition. To use dual-channel waveforms, two files holding single-channel waveform can be loaded separately:

```
wave w1 = 'wave1';
wave w2 = 'wave2';

playWave(1,w1, 2,w2);
```

The file must use the extension ".wave". Each sample is a word of 16 bits little-endian. The bits are assigned as follow:

- Bit 15-2: Wave, as 14 bit signed integer
- Bit 1: Marker 2
- Bit 0: Marker 1

As example, the waveform [-1.0, 0.0, 1.0], without marker will be saved as **04 80 00 00 FC 7F**. The same waveform with marker1 [1,0,0] and marker2 [1,1,0] would be **FD 7F 01 00 02 00**.

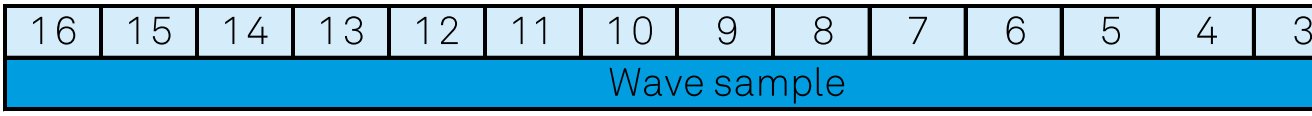


Figure 3.7. Binary format of samples

Note

By default the AWG compiler loads in memory all the binary waveforms present in the **directory**. To achieve faster compilation, it's advised to keep there only the required waveforms or to use the **compiler/waveform**.

ASCII CSV

As alternative to binary waveform, the ASCII CSV format can be used. The CSV file should contain floating-point values in the range from –1.0 to +1.0 and contain one or two columns, corresponding to the single- and dual-channel waveforms. The file must use the extension ".csv". As an example, the following specify a dual-channel wave with a length of 16 samples:

```
-1.0   0.0
-0.8   0.0
-0.7   0.1
-0.5   0.2
-0.2   0.3
-0.1   0.2
 0.1   0.0
 0.2  -0.1
 0.7  -0.3
 1.0  -0.2
 0.9  -0.3
 0.8  -0.2
 0.4  -0.1
 0.0  -0.1
-0.5  -0.1
-0.8   0.0
```

In order to obtain digital marker data from a file, specify a second wave file with integer instead of floating-point values. The marker bits are encoded in the binary representation of the integer (i.e., integer 1 corresponds to the first marker high, 2 corresponds to the second marker high, and 3 corresponds to both bits high). Later in the program add up the analog and the marker waveforms. For instance, if the floating-point analog data are contained in **wave_file_analog.csv** and the integer marker data in **wave_file_digital.csv**, the following code can be used to combine and play them.

```
wave w_analog  = "wave_file_analog";
wave w_digital = "wave_file_digital";
wave w = w_analog + w_digital;
playWave(w);
```

As an alternative to specifying analog data as floating-point values in one file, and marker data as integer values in a second file, both may be combined into one file containing integer values. The integer values in that file should be 18-bit unsigned integers with the two least significant bits being the markers. The values are mapped to 0 $\Rightarrow$ -FS, 262143 $\Rightarrow$ +FS, with FS equal to the full scale. This integer version of the CSV format is not to be confused with the binary file format documented previously. To optimize compilation speed, the binary format should be preferred over both versions of the CSV format.

3.3.4. HDAWG Channel Grouping

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

0: Use the outputs in groups of 2; each sequencer program controls 2 outputs. Each group n=0,1,2,3 of AWGs (respectively n=0,1 on HDAWG4 instruments) is configured by the /DEV... ./AWGS/n branches. Each of these 4 groups requires its own instance of the AWG Module and `index` should be set to n=0,1,2,3 for each group accordingly.

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

2: HDAWG8 devices only. Use the outputs in a single group of 8; the (single) sequencer program controls 8 outputs. There is only one group (n=0) of 8 AWGs which is configured by the / DEV..../AWGS/0 branch. Only one instance of the AWG Module is required and its value of `index` should be 0.

Table 3.1. Overview of the device nodes and the value of `index` used indifferent channel grouping configurations on HDAWG8 instruments.

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

Table 3.2. Overview of the device nodes and the value of `index` used indifferent channel grouping configurations on HDAWG4 instruments.

Value of CHANNELGROUPING	Number of Cores	AWG Core	Corresponding device AWG branch index	Value of `index`
0	2	1	/DEV..../AWGS/0	0
		2	/DEV..../AWGS/1	1

Value of CHANNELGROUPING	Number of Cores	AWG Core	Corresponding device AWG branch index	Value of `index`
1	1	1	/DEV..../AWGS/0	0

3.3.5. AWG Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the AWG module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

awg

//awg/enable

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Start the AWG sequencers. In MDS mode, this will enable all devices in the correct order.

compiler

//compiler/sourcefile

Properties: Read, Write
Type: String
Unit: None

The filename of an AWG sequencer program file to compile and load. The file must be located in the "awg/src" sub-directory of the LabOne user directory. This directory path is provided by the value of the read-only directory parameter.

//compiler/sourcestring

Properties: Read, Write
Type: String
Unit: None

A string containing an AWG sequencer program may directly loaded to this parameter using the module command setString. This allows compilation and upload of a sequencer program without saving it to a file first. Compilation starts automatically after compiler/sourcestring has been set.

//compiler/start

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to start compiling the AWG sequencer program specified by compiler/ sourcefile. The module sets compiler/ start to 0 once compilation has successfully completed (or failed). If compiler/upload is enabled then the sequencer program will additionally be uploaded to the AWG upon after successful compilation.

//compiler/status

Properties:	Read
Type:	Integer (enumerated)
Unit:	None

Compilation status

-1	Idle.
0	Compilation successful.
1	Compilation failed.
2	Compilation completed with warnings.

//compiler/statusstring

Properties:	Read
Type:	String
Unit:	None

Status message of the compiler.

//compiler/upload

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Specify whether the sequencer program should be automatically uploaded to the AWG following successful compilation.

//compiler/waveforms

Properties:	Read, Write

Type:	String
Unit:	None

A comma-separated list of waveform CSV files to be used by the AWG sequencer program.

device

//device

Properties:	Read, Write
Type:	String
Unit:	None

The target device for AWG sequencer programs upload, e.g. 'dev2006'.

directory

//directory

Properties:	Read, Write
Type:	String
Unit:	None

The path of the LabOne user directory. The AWG Module uses the following subdirectories in the LabOne web server directory: "awg/src": Contains AWG sequencer program source files (user created); "awg/elf": Contains compiled AWG binary (ELF) files (created by the module); "awg/waves": Contains CSV waveform files (user created).

elf

//elf/checksum

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

The checksum of the generated ELF file.

//elf/file

Properties:	Read, Write
Type:	String
Unit:	None

The filename of the compiled binary ELF file. If not set, the name is automatically set based on the source filename. The ELF file will be saved by the AWG Module in the "awg/elf" sub-directory of the LabOne user directory. This directory path is provided by the value of the read-only directory parameter.

//elf/status

Properties:	Read
Type:	Integer (enumerated)
Unit:	None

Status of the ELF file upload.

-1	Idle.
0	Upload successful.
1	Upload failed.
2	Upload in progress.

//elf/upload

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to start uploading the AWG sequencer program to the device. The module sets elf/upload to 0 once the upload has finished.

index

//index

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The index of the current AWG Module to use when running with multiple AWG groups. See section on channel grouping in the manual for further explanation.

mds

//mds/group

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The MDS group (multiDeviceSyncModule/group) to use for synchronized AWG playback.

progress

//progress

Properties:	Read
Type:	Double
Unit:	None

Reports the progress of the upload as a value between 0 and 1.

sequencertype

//sequencertype

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Type of sequencer to compile for. For all devices but the SHFQC, the sequencer type is deduced from the device type, and this node is ignored. For the SHFQC, the sequencer type must be defined ("qa" or "sg").

0 `auto-detect`
The sequencer type is deduced from the device type (for all devices but the SHFQC).

1 `qa`
QA sequencer

2 `sg`
SG sequencer

3.4. Data Acquisition Module

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

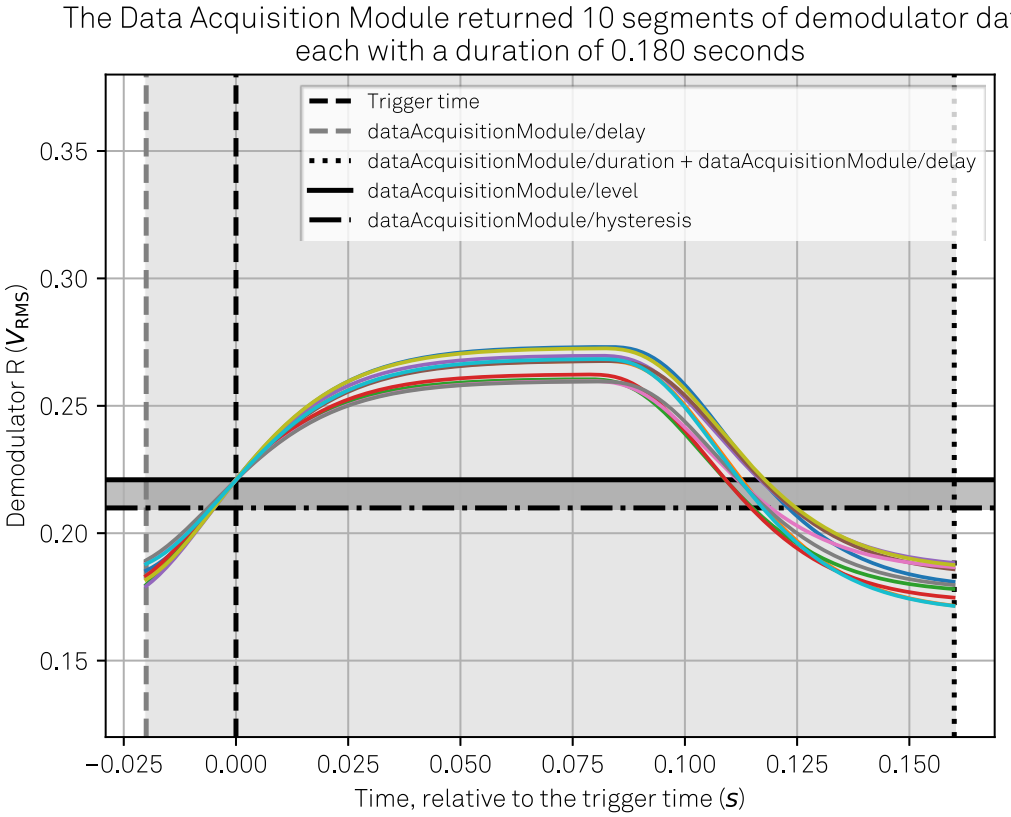


Figure 3.8. The plot was generated by `example_data_acquisition_edge.py`, an example distributed on our public GitHub repository (https://github.com/zhinst/labone-api-examples). The plot shows 10 bursts of data acquired from a demodulator; each burst was recorded when the demodulator's R value exceeded a specified threshold using a positive edge trigger.

3.4.1. DAQ Module Acquisition Modes and Trigger Types

This section lists the required parameters and special considerations for each trigger mode. For reference documentation of the module's parameters please see Section 3.4.3.

Table 3.3. Overview of the acquisition modes available in the Data Acquisition Module.

Mode / Trigger Type	Description	Value of `type`
Continuous	Continuous recording of data.	0
Edge	Edge trigger with noise rejection.	1
Pulse	Pulse width trigger with noise rejection.	3

Mode / Trigger Type	Description	Value of **type**
Tracking (Edge or Pulse)	Level tracking trigger to compensate for signal drift.	4
Digital	Digital trigger with bit masking.	2
Hardware	Trigger on one of the instrument's hardware trigger channels (not available on HF2).	6
Pulse Counter	Trigger on the value of an instrument's pulse counter (requires CNT Option).	8

Continuous Acquisition

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

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

Acquisition using Level Edge Triggering

Parameters specific to edge triggering are:

- **level**,
- **hysteresis**.

The user can request automatic calculation of the **level** and **hysteresis** parameters by setting the **findlevel** parameter to 1. Please see Determining the Trigger Level automatically for more information.

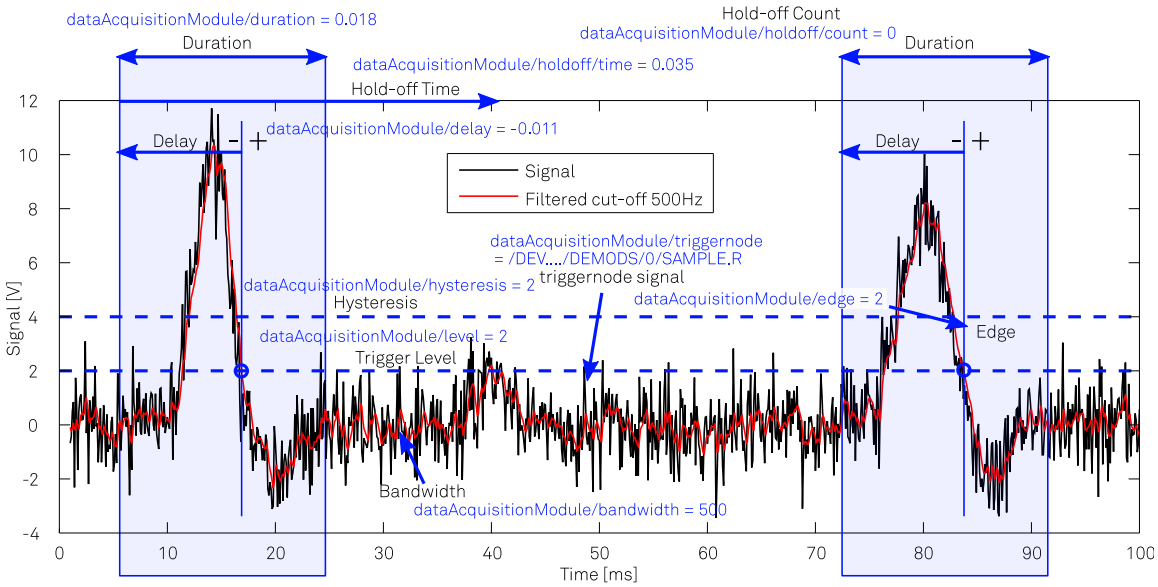


Figure 3.9. Explanation of the Data Acquisition Module's parameters for an Edge Trigger.

Acquisition using Pulse Triggering

Parameters specific to pulse triggering are:

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

The user can request automatic calculation of the `level` and `hysteresis` parameters by setting the `findlevel` parameter to 1. Please see Determining the Trigger Level automatically for more information.

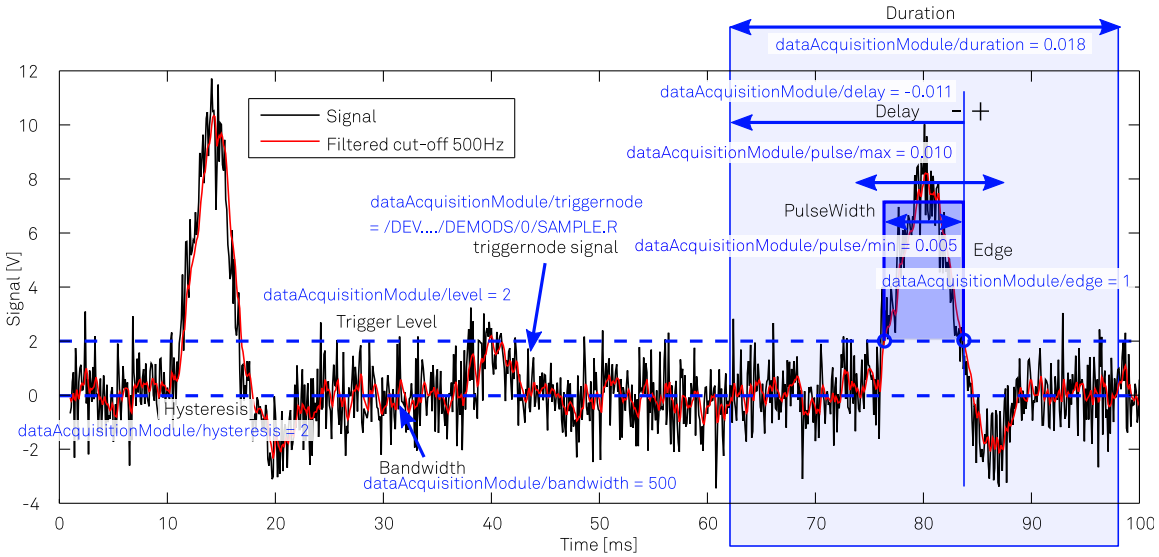


Figure 3.10. Explanation of the Data Acquisition Module's parameters for a positive Pulse Trigger.

Acquisition using Tracking Edge or Pulse Triggering

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

- `bandwidth`

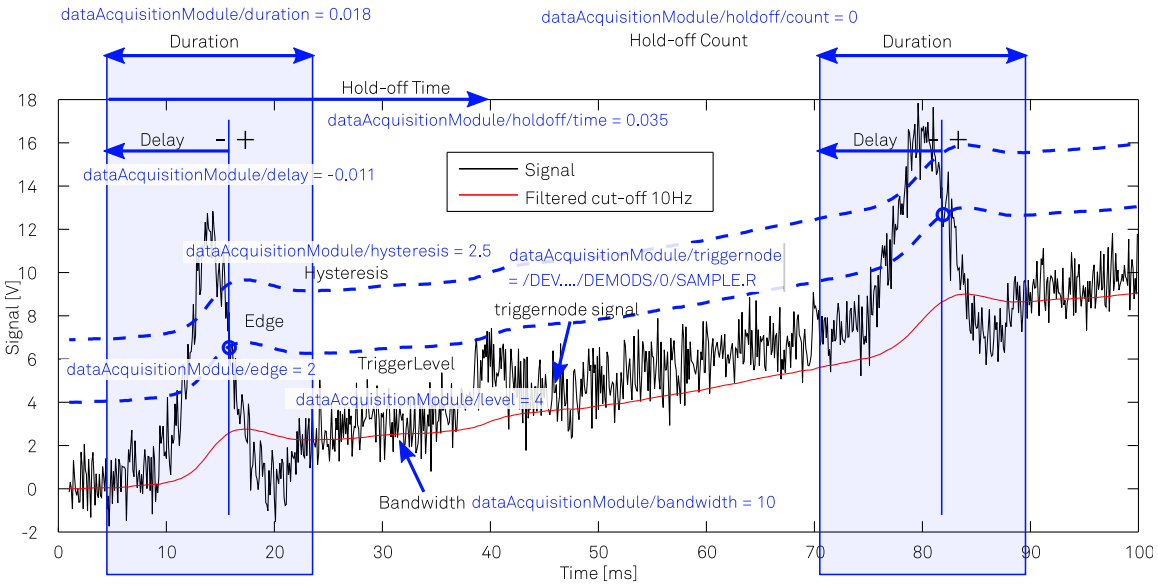


Figure 3.11. Explanation of the Data Acquisition Module's parameters for a Tracking Trigger.

Acquisition using Digital Triggering

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

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

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

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

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

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

Acquisition using Hardware Triggering

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

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

The hardware trigger type is not supported on HF2 instruments.

Acquisition using Pulse Counter Triggering

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

- eventcount/mode.

The **triggernode** must be configured to be a pulse counter sample:

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

Determining the Trigger Level automatically

The Data Acquisition Module can calculate the **level** and **hysteresis** parameters based on the current input signal for edge, pulse, tracking edge and tracking pulse trigger types. This is particularly useful when using a tracking trigger, where the trigger level is relative to the output of the low-pass filter tracking the input signal's average (see Figure 3.11). In the LabOne User Interface this functionality corresponds to the "Find" button in the Settings sub-tab of the Data Acquisition Tab.

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

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

```
level = trigger.getDouble('level')
hysteresis = trigger.getDouble('hysteresis')
```

Example 3.1. Python code demonstrating how to use the `findlevel` parameter. Taken from the Python example example_data_acquisition_grid.

3.4.2. Signal Subscription

The Data Acquisition Module uses dot notation for subscribing to the signals. Whereas with the Software Trigger (Recorder Module) you subscribe to an entire streaming node, e.g. /DEV... ./DEMODS/N/SAMPLE and get all the signal components of this node back, with the Data Acquisition Module you specify the exact signal you are interested in capturing, e.g. /DEV..../DEMODS/N/SAMPLE.r /DEV..../DEMOD/0/SAMPLE.phase. In addition, by appending suffixes to the signal path, various operations can be applied to the source signal and cascaded to obtain the desired result. Some examples are given below (the /DEV.../DEMODS/n/SAMPLE prefix has been omitted):

`x`	Demodulator sample x component.
`r.avg`	Average of demodulator sample abs(x + iy).
`x.std`	Standard deviation of demodulator sample x component.
`xiy.fft.abs.std`	Standard deviation of complex FFT of x + iy.
`phase.fft.abs.avg`	Average of real FFT of linear corrected phase.
`freq.fft.abs.pwr`	Power of real FFT of frequency.
`r.fft.abs`	Real FFT of abs(x + iy).
`df.fft.abs`	Real FFT of demodulator phase derivative (dθ/dt)/(2π).
`xiy.fft.abs.pwr`	Power of complex FFT of x + iy.
`xiy.fft.abs.filter`	Demodulator low-pass filter transfer function. Divide xiy.fft.abs by this to obtain a compensated FFT.

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

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

Signal Subscription Options

source signal			
Node	Signal Name	Description (Path of the node containing the signal(s))	Comment
demod	x	Demodulator output in-phase component	
	y	Demodulator output quadrature component	
	r	Demodulator output amplitude	
	theta	Demodulator output phase	
	frequency	Oscillator frequency	
	auxin0	Auxilliary input channel 1	
	auxin1	Auxilliary input channel 2	

source signal			
	xiy	Combined demodulator output in-phase and quadrature components	complex output (can only be used as FFT input))
	df	Demodulator output phase derivative (can only be used for FFT(dθ/dt)/(2π)	
impedance	realz	In-phase component of impedance sample	
	imagz	Quadrature component of impedance sample	
	absz	Amplitude of impedance sample	
	phasez	Phase of impedance sample	
	frequency	Oscillator frequency	
	param0	Measurement parameter that depends on circuit configuration	
	param1	Measurement parameter that depends on circuit configuration	
	drive	Amplitude of the AC signal applied to the device under test	
	bias	DC Voltage applied to the device under test	
	z	Combined impedance in-phase and quadrature components	complex (can only be used as FFT input)
other		Nodes not listed here	Nodes containing only one signal do not have a source_signal field.

FFT (optional)		
Name (node_path)	**Description (Path of the node containing the signal(s))**	**Comment**
FFT		complex output

complex_selector (mandatory with FFT)		
Name (node_path)	**Description (Path of the node containing the signal(s))**	**Comment**
real	Real component of FFT	
imag	Imaginary component of FFT	
abs	Absolute component of FFT	
phase	Phase component of FFT	

filter (optional)		
Name (node_path)	**Description (Path of the node containing the signal(s))**	**Comment**
filter	Helper signal representing demodulator low-pass filter transfer function. It can only be applied to 'abs' FFT output of	

filter (optional)		
	complex demodulator source signal, i.e. 'xiy.fft.abs.filter'. No additional operations are permitted. Can be used to compensate the FFT result for the demodulator low-pass filter.	

pwr (optional)		
Name (node_path)	**Description (Path of the node containing the signal(s))**	**Comment**
pwr	Power calculation	

math_operation (optional)		
Name (node_path)	**Description (Path of the node containing the signal(s))**	**Comment**
avg	Average of grid repetitions (parameter grid/repetitions)	
std	Standard deviation	

3.4.3. Data Acquisition Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the data acquisition module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

awgcontrol

//awgcontrol

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable interaction with the AWG. If enabled, the row number is identified based on the digital row ID number set by the AWG. If disabled, every new trigger event is attributed to a new row sequentially.

bandwidth

//bandwidth

Properties: Read, Write
Type: Double
Unit: Hz

Set to a value other than 0 in order to apply a low-pass filter with the specified bandwidth to the triggernode signal before applying the trigger criteria. For edge and pulse trigger use a bandwidth larger than the trigger signal's sampling rate divided by 20 to keep the phase delay. For tracking filter use a bandwidth smaller than the trigger signal's sampling rate divided by 100 to track slow signal components like drifts. The value of the filtered signal is returned by read() under the path /DEV..../TRIGGER/LOWPASS.

bitmask

//bitmask

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Specify a bit mask for the DIO trigger value. The trigger value is bits AND bit mask (bitwise). Only used when the trigger type is digital.

bits

//bits

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Specify the value of the DIO to trigger on. All specified bits have to be set in order to trigger. Only used when the trigger type is digital.

buffercount

//buffercount

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

The number of buffers used internally by the module for data recording.

buffersize

//buffersize

Properties:	Read

Type:	Double
Unit:	Seconds

The buffersize of the module's internal data buffers.

clearhistory

/clearhistory

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to clear all the acquired data from the module. The module immediately resets clearhistory to 0 after it has been set to 1.

count

/count

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The number of trigger events to acquire in single-shot mode (when endless is set to 0).

delay

/delay

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time delay of trigger frame position (left side) relative to the trigger edge. delay=0: Trigger edge at left border; delay<0: trigger edge inside trigger frame (pretrigger); delay>0: trigger edge before trigger frame (posttrigger)

device

/device

Properties:	Read, Write

Type: String
Unit: None

The device serial to be used with the Data Acquisition Module, e.g. dev123 (compulsory parameter).

duration

//duration

Properties: Read, Write
Type: Double
Unit: Seconds

The recording length of each trigger event. This is an input parameter when the sampling mode (grid/mode) is either nearest or linear interpolation. In exact sampling mode duration is an output parameter; it is calculated and set by the module based on the value of grid/cols and the highest rate of all the subscribed signal paths.

edge

//edge

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

The trigger edge to trigger upon when running a triggered acquisition mode.

1 `rising`
Rising edge
2 `falling`
Falling edge
3 `both`
Both rising and falling

enable

//enable

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Set to 1 to enable the module and start data acquisition (is equivalent to calling execute()).

endless

//endless

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to enable endless triggering. Set to 0 and use count if the module should only acquire a certain number of trigger events.

eventcount

//eventcount/mode

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Specifies the trigger mode when the triggernode is configured as a pulse counter sample value (/DEV..../CNTS/0/SAMPLE.value).

0 `every_sample`
Trigger on every sample from the pulse counter, regardless of the counter value.

1 `incrementing_counter`
Trigger on incrementing counter values.

fft

//fft/absolute

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to shift the frequencies in the FFT result so that the center frequency becomes the demodulation frequency rather than 0 Hz (when disabled).

//fft/window

Properties:	Read, Write
Type:	Integer (enumerated)

Unit: None

The FFT window function to use (default 1 = Hann). Depending on the application, it makes a huge difference which of the provided window functions is used. Please check the literature to find out the best trade off for your needs.

0 `rectangular`
Rectangular

1 `hann`
Hann

2 `hamming`
Hamming

3 `blackman_harris`
Blackman Harris 4 term

16 `exponential`
Exponential (ring-down)

17 `cos`
Cosine (ring-down)

18 `cos_squared`
Cosine squared (ring-down)

findlevel

//findlevel

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Set to 1 to automatically find appropriate values of the trigger level and hysteresis based on the current triggernode signal value. The module sets findlevel to 0 once the values have been found and set.

flags

//flags

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Record flags. FILL = 0x1: always enabled; ALIGN = 0x2: always enabled; THROW = 0x4: Throw if sample loss is detected; DETECT = 0x8: always enabled.

forcetrigger

//forcetrigger

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to force acquisition of a single trigger for all subscribed signal paths (when running in a triggered acquisition mode). The module immediately resets forcetrigger to 0 after it has been set to 1.

grid

//grid/cols

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Specify the number of columns in the returned data grid (matrix). The data along the horizontal axis is resampled to the number of samples defined by grid/cols. The grid/mode parameter specifies how the data is sample onto the time, respectively frequency, grid.

//grid/direction

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

The direction to organize data in the grid's matrix.

0 `forward`

Forward. The data in each row is ordered chronologically, e.g., the first data point in each row corresponds to the first timestamp in the trigger data.

1 `reverse`

Reverse. The data in each row is in reverse chronological order, e.g., the first data point in each row corresponds to the last timestamp in the trigger data.

2 `bidirectional`

Bidirectional. The ordering of the data alternates between Forward and Backward ordering from row-to-row. The first row is Forward ordered.

//grid/mode

Properties:	Read, Write
Type:	Integer (enumerated)

Unit: None

Specify how the acquired data is sampled onto the matrix's horizontal axis (time or frequency). Each trigger event becomes a row in the matrix and each trigger event's subscribed data is sampled onto the grid defined by the number of columns (grid/cols) and resampled as specified with this parameter.

1 `nearest`

Use the closest data point (nearest neighbour interpolation).

2 `linear`

Use linear interpolation.

4 Do not resample the data from the subscribed signal path(s) with the highest sampling rate; the horizontal axis data points are determined from the sampling rate and the value of grid/cols. Subscribed signals with a lower sampling rate are upsampled onto this grid using linear interpolation.

/grid/overwrite

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

If enabled, the module will return only one data chunk (grid) when it is running, which will then be overwritten by subsequent trigger events.

/grid/repetitions

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Number of statistical operations performed per grid. Only applied when the subscribed signal path is, for example, an average or a standard deviation.

/grid/rowrepetition

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable row-wise repetition. With row-wise repetition, each row is calculated from successive repetitions before starting the next row. With grid-wise repetition, the entire grid is calculated with each repetition.

/grid/rows

Properties: Read, Write

Type:	Integer (64 bit)
Unit:	None

Specify the number of rows in the grid's matrix. Each row is the data recorded from one trigger event.

//grid/waterfall

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to enable waterfall mode: Move the data upwards upon each trigger event; the data from newest trigger event is placed in row 0.

historylength

//historylength

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Sets an upper limit for the number of data captures stored in the module.

holdoff

//holdoff/count

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The number of skipped trigger events until the next trigger event is acquired.

//holdoff/time

Properties:	Read, Write
Type:	Double
Unit:	Seconds

The hold-off time before trigger acquisition is re-armed again. A hold-off time smaller than the duration will produce overlapped trigger frames.

hysteresis

//hysteresis

Properties:	Read, Write
Type:	Double
Unit:	Many

If non-zero, hysteresis specifies an additional trigger criteria to level in the trigger condition. The trigger signal must first go higher, respectively lower, than the hysteresis value and then the trigger level for positive, respectively negative edge triggers. The hysteresis value is applied below the trigger level for positive trigger edge selection. It is applied above for negative trigger edge selection, and on both sides for triggering on both edges. A non-zero hysteresis value is helpful to trigger on the correct edge in the presence of noise to avoid false positives.

level

//level

Properties:	Read, Write
Type:	Double
Unit:	Many

The trigger level value.

preview

//preview

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If set to 1, enable the data of an incomplete trigger to be read. Useful for long trigger durations (or FFTs) by providing access to the intermediate data.

pulse

//pulse/max

Properties:	Read, Write
Type:	Double
Unit:	Seconds

The maximum pulse width to trigger on when using a pulse trigger.

/pulse/min

Properties:	Read, Write
Type:	Double
Unit:	Seconds

The minimum pulse width to trigger on when using a pulse trigger.

refreshrate

/refreshrate

Properties:	Read, Write
Type:	Double
Unit:	Hz

Set the maximum refresh rate of updated data in the returned grid. The actual refresh rate depends on other factors such as the hold-off time and duration.

save

/save/csvlocale

Properties:	Read, Write
Type:	String
Unit:	None

The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C": Dot for the decimal point and no digit grouping (default); " (empty string): Use the symbols set in the language and region settings of the computer.

/save/csvseparator

Properties:	Read, Write
Type:	String
Unit:	None

The character to use as CSV separator when saving files in this format.

/save/directory

Properties:	Read, Write
Type:	String

Unit: None

The base directory where files are saved.

/save/fileformat

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

The format of the file for saving data.

0 `mat`
MATLAB
1 `csv`
CSV
2 `zview`
ZView (Impedance data only)
3 `sxm`
SXM (Image format)
4 `hdf5`
HDF5

/save/filename

Properties: Read, Write
Type: String
Unit: None

Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g. daq_000.

/save/save

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Initiate the saving of data to file. The saving is done in the background. When the save has finished, the module resets this parameter to 0.

/save/saveonread

Properties: Read, Write
Type: Integer (64 bit)

Unit: None

Automatically save the data to file immediately before reading out the data from the module using the read() command. Set this parameter to 1 if you want to save data to file when running the module continuously and performing intermediate reads.

spectrum

//spectrum/autobandwidth

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Set to 1 to initiate automatic adjustment of the subscribed demodulator bandwidths to obtain optimal alias rejection for the selected frequency span which is equivalent to the sampling rate. The FFT mode has to be enabled (spectrum/enable) and the module has to be running for this function to take effect. The module resets spectrum/autobandwidth to 0 when the adjustment has finished.

//spectrum/enable

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enables the FFT mode of the data Acquisition module, in addition to time domain data acquisition. Note that when the FFT mode is enabled, the grid/cols parameter value is rounded down to the nearest binary power.

//spectrum/frequencyspan

Properties: Read, Write
Type: Double
Unit: None

Sets the desired frequency span of the FFT.

//spectrum/overlapped

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enables overlapping FFTs. If disabled (0), FFTs are performed on distinct abutting data sets. If enabled, the data sets of successive FFTs overlap based on the defined refresh rate.

triggered

//triggered

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

Indicates whether the module has recently triggered: 1=Yes, 0=No.

triggernode

//triggernode

Properties:	Read, Write
Type:	String
Unit:	None

The node path and signal that should be used for triggering, the node path and signal should be separated by a dot (.), e.g. /DEV.../DEMODS/0/SAMPLE.X.

type

//type

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Specifies how the module acquires data.

0 `continuous`
Continuous acquisition (trigger off).

1 `analog_edge_trigger`
Analog edge trigger.

2 `digital_trigger`
Digital trigger mode (on DIO source).

3 `analog_pulse_trigger`
Analog pulse trigger.

4 `analog_tracking_trigger`
Analog tracking trigger.

5 `change_trigger`
Change trigger.

6 `hardware_trigger`

Hardware trigger (on trigger line source).

7 `pulse_tracking_trigger`

Pulse tracking trigger, see also bandwidth.

8 `event_count_trigger`

Event count trigger (on pulse counter source).

3.5. 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 synchronously, i.e., these functions block until loading or saving has completed, the desired behavior in most cases. Advanced users can use the Device Settings module directly if they need to implement loading or saving asynchronously (non-blocking).

3.5.1. Device Settings Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the device settings module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

command

//command

Properties:	Read, Write
Type:	String
Unit:	None

The command to execute: 'save' = Read device settings and save to file; 'load' = Load settings from file an write to device; 'read' = Read device settings only (no save).

device

//device

Properties:	Read, Write
Type:	String
Unit:	None

Comma separated list of devices that should be used for loading/saving device settings, e.g. 'dev99,dev100'.

errortext

//errortext

Properties:	Read

Type:	String
Unit:	None

The error text used in error messages.

filename

/filename

Properties:	Read, Write
Type:	String
Unit:	None

Name of settings file to use.

finished

/finished

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

The status of the command.

path

/path

Properties:	Read, Write
Type:	String
Unit:	None

Directory where settings files should be located. If not set, the default settings location of the LabOne software is used.

throwonerror

/throwonerror

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Throw an exception is there was error executing the command.

3.6. Impedance Module

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

3.6.1. Impedance Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the impedance module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

calibrate

//calibrate

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If set to true will execute a compensation for the specified compensation condition.

comment

//comment

Properties:	Read, Write
Type:	String
Unit:	None

Comment string that will be saved together with the compensation data.

device

//device

Properties:	Read, Write
Type:	String
Unit:	None

Device string defining the device on which the compensation is performed.

directory

//directory

Properties:	Read, Write
Type:	String
Unit:	None

The directory where files are saved.

expectedstatus

//expectedstatus

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Bit field of the load condition that the corresponds a full compensation. If status is equal the expected status the compensation is complete.

filename

//filename

Properties:	Read, Write
Type:	String
Unit:	None

The name of the file to use for user compensation.

freq

//freq/samplecount

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Number of samples of a compensation trace.

//freq/start

Properties:	Read, Write

Type:	Double
Unit:	Hz

Start frequency of compensation traces.

//freq/stop

Properties:	Read, Write
Type:	Double
Unit:	Hz

Stop frequency of compensation traces.

highimpedanceload

//highimpedanceload

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable a second high impedance load compensation for the low current ranges.

load

//load

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Load the impedance user compensation data from the file specified by filename.

loads

//loads/0/c

Properties:	Read, Write
Type:	Double
Unit:	F

Parallel capacitance of the first compensation load (SHORT).

//loads/0/r

Properties: Read, Write
Type: Double
Unit: Ohm

Resistance value of first compensation load (SHORT).

//loads/1/c

Properties: Read, Write
Type: Double
Unit: F

Parallel capacitance of the first compensation load (SHORT).

//loads/1/r

Properties: Read, Write
Type: Double
Unit: Ohm

Resistance value of first compensation load (SHORT).

//loads/2/c

Properties: Read, Write
Type: Double
Unit: F

Parallel capacitance of the first compensation load (SHORT).

//loads/2/r

Properties: Read, Write
Type: Double
Unit: Ohm

Resistance value of first compensation load (SHORT).

//loads/3/c

Properties: Read, Write

Type: Double
Unit: F

Parallel capacitance of the first compensation load (SHORT).

//loads/3/r

Properties: Read, Write
Type: Double
Unit: Ohm

Resistance value of first compensation load (SHORT).

message

//message

Properties: Read
Type: String
Unit: None

Message string containing information, warnings or error messages during compensation.

mode

//mode

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

Compensation mode to be used. Defines which load steps need to be compensated.

0 `none`
None
1 `short`
S (Short)
4 `load`
L (Load)
5 `short_load`
SL (Short-Load)
8 `load_load_load`
LLL (Load-Load-Load)

openstep

//openstep

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Perform an additional open compensation step.

path

//path

Properties:	Read, Write
Type:	String
Unit:	None

The path of the directory where the user compensation file is located.

precision

//precision

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Precision of the compensation. Will affect time of a compensation and reduces the noise on compensation traces.

0	`standard` Standard speed
1	`high` Low speed / high precision

progress

//progress

Properties:	Read
Type:	Double

Unit: None

Progress of a compensation condition.

save

/save

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Save the current impedance user compensation data to the file specified by filename.

status

/status

Properties: Read
Type: Integer (64 bit)
Unit: None

Bit coded field of the already compensated load conditions (bit 0 = first compensation step, bit 1 = second compensation step, ...).

step

/step

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

Compensation step to be performed when calibrate indicator is set to true.

0 `first_load`
First load
1 `second_load`
Second load
2 `third_load`
Third load
3 `fourth_load`
Fourth load

todevice

//todevice

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled will automatically transfer compensation data to the persistent flash memory in case of a valid compensation.

validation

//validation

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the validation of compensation data. If enabled the compensation is checked for too big deviation from specified load.

3.7. Multi-Device Synchronisation Module

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

3.7.1. Multi-Device Synchronisation Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the multi-device synchronisation module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

devices

//devices

Properties:	Read, Write
Type:	String
Unit:	None

Defines which instruments should be included in the synchronization. Expects a comma-separated list of devices in the order the devices are connected.

group

//group

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Defines in which synchronization group should be accessed by the module.

message

//message

Properties:	Read
Type:	String
Unit:	None

Status message of the module.

phasesync

//phasesync

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to reset the phases of all oscillators on all of the synchronized devices.

recover

//recover

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to resynchronize the device group if the synchronization has been lost.

start

//start

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to true to start the synchronization process.

status

//status

Properties:	Read
Type:	Integer (enumerated)
Unit:	None

Status of the synchronization process.

-1 error

0	idle
1	synchronization in progress
2	successful synchronization

3.8. PID Advisor Module

The PID Advisor Module provides the functionality available in the Advisor, Tuner and Display sub-tabs of the LabOne User Interface's PID / PLL tab. The PID Advisor is a mathematical model of the instrument's PID and can be used to calculate PID controller parameters for optimal feedback loop performance. The controller gains calculated by the module can be easily transferred to the device via the API and the results of the Advisor's modeling are available as Bode and step response plot data as shown in Figure 3.12.

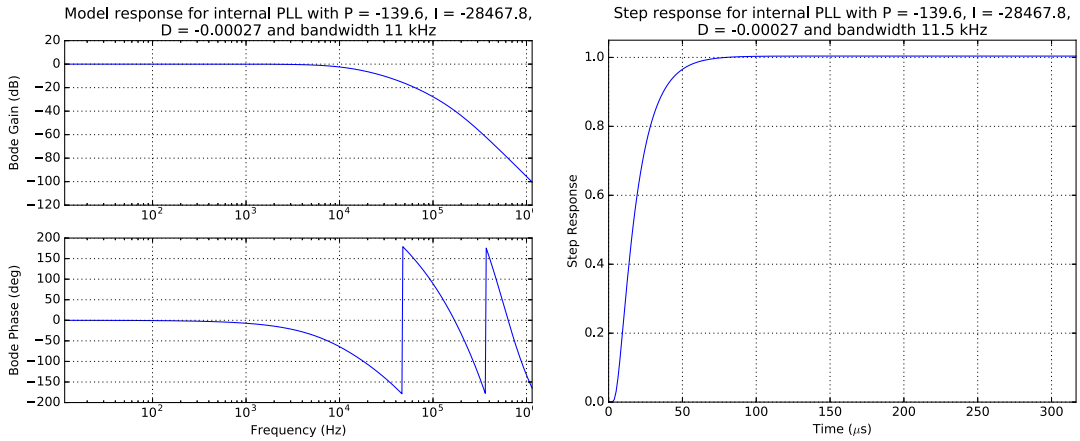


Figure 3.12. The plot was generated by `example_pid_advisor_pll.py`, an example distributed on our public GitHub repository (https://github.com/zhinst/labone-api-examples). It configures the PID Advisor to optimize an internal PLL control loop. The data used in the Bode and step response plots is returned by the PID Advisor in the `bode` respectively `step` output parameters.

3.8.1. PID Advisor Module Work-Flow

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

1. Create an instance of the PID Advisor module.
2. Configure the module's parameters using **`set()`** to specify, for example, which of the instrument's PIDs to use and which type of device under test (DUT) to model. If values are specified for the P, I and D gains they serve as initial values for the optimization process. See Section 3.8.5 for a full list of PID Advisor parameters.
3. Start the module by calling **`execute()`**.
4. Start optimization of the PID parameters by setting the **`calculate`** parameter to 1. The optimization process has finished when the value of **`calculate`** returns to 0. Optimization may take up to a minute to complete, but is much quicker in most cases.
5. Read out the optimized parameters, Bode and step response plot data (see Figure 3.12) for inspection using **`get()`**.
6. Transfer the optimized gain parameters from the Advisor Module to the instrument's nodes by setting the **`todevice`** parameter to 1.
7. Enable the instrument's PID (/DEV..../PIDS/n/ENABLE).
8. The Auto Tune functionality may be additionally enabled by setting **`tune`** to 1 and configuring the parameters in the **`tuner`** branch. This functionality continuously updates the

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

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

3.8.2. PLL Parameter Optimization on HF2 Instruments

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

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

3.8.3. Instrument Settings written by todevice

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

For HF2 instruments there are two main cases to differentiate between, defined by whether **type** is set to "pid" or "pll" (see Section 3.8.2 for an explanation). For UHF and MF devices **type** can only be set to "pid", for these devices Table 3.4 and Table 3.5 describe which device nodes are configured.

Table 3.4. The device nodes configured when type is "pid" (default behavior).The value of n in device nodes corresponds to the value of index.

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

Table 3.5. The additional device nodes configured when type is "pid" (default behavior) and dut/source=4 (internal PLL). The value of n in device nodes corresponds to the value of index.

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

Device node (/DEV.../)	Value set	Device class
PIDS/n/LIMITLOWER	Calculated **-bw*2**, if **autolimit=1**.	Not HF2
PIDS/n/LIMITUPPER	Calculated **bw*2**, if **autolimit=1**.	Not HF2
PIDS/n/RANGE	Calculated **bw*2**, if **autolimit=1**.	HF2 only

Table 3.6. The device nodes configured when `type` is "pll" (HF2 instruments only - see Section 3.8.2 for an explanation). The value of n in device nodes corresponds to the value of `index`.

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

3.8.4. Monitoring the PID's Output

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

MF and UHF Instruments

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

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

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

HF2 Instruments

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

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

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

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

The PLL Module

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

3.8.5. PID Advisor Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the PID advisor module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

advancedmode

//advancedmode

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled, automatically calculate the start and stop value used in the Bode and step response plots.

auto

//auto

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled, automatically trigger a new optimization process upon an input parameter value change.

bode

//bode

Properties:	Read
Type:	ZIAdvisorWave
Unit:	None

Contains the resulting bode plot of the PID simulation.

bw

//bw

Properties:	Read
Type:	Double
Unit:	Hz

Calculated system bandwidth.

calculate

//calculate

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Set to 1 to start the PID Advisor's modelling process and calculation of optimal parameters. The module sets calculate to 0 when the calculation is finished.

demod

//demod/harmonic

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output. Specifies the demodulator's harmonic to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV..../DEMODS/m/HARMONIC) when the PID is enabled.

//demod/order

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output. Specifies the demodulator's order to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV..../DEMODS/m/ORDER) when the PID is enabled.

//demod/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Only relevant when /DEV.../PIDS/n/INPUT is configured to be a demodulator output and pidAdvisor/pid/autobw=0. Specify the demodulator's timeconstant to use in the PID Advisor model. This value will be transferred to the instrument node (/DEV..../DEMODS/m/TIMECONSTANT) when the PID is enabled.

device

//device

Properties:	Read, Write
Type:	String
Unit:	None

Device string specifying the device for which the PID advisor is performed.

display

//display/freqstart

Properties:	Read, Write
Type:	Double
Unit:	Hz

Start frequency for Bode plot. If advancedmode=0 the start value is automatically derived from the system properties.

//display/freqstop

Properties:	Read, Write

Type:	Double
Unit:	Hz

Stop frequency for Bode plot.

/display/timestart

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Start time for step response. If advancedmode=0 the start value is 0.

/display/timestop

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Stop time for step response.

dut

/dut/bw

Properties:	Read, Write
Type:	Double
Unit:	Hz

Bandwidth of the DUT (device under test).

/dut/damping

Properties:	Read, Write
Type:	Double
Unit:	None

Damping of the second order low pass filter.

/dut/delay

Properties:	Read, Write
Type:	Double
Unit:	Seconds

IO Delay of the feedback system describing the earliest response for a step change.

//dut/fcenter

Properties:	Read, Write
Type:	Double
Unit:	Hz

Resonant frequency of the of the modelled resonator.

//dut/gain

Properties:	Read, Write
Type:	Double
Unit:	Depends on Input, Output, and DUT model

Gain of the DUT transfer function.

//dut/q

Properties:	Read, Write
Type:	Double
Unit:	None

Quality factor of the modelled resonator.

//dut/source

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Specifies the model used for the external DUT (device under test) to be controlled by the PID.

1 `low_pass_1st_order`
Low-pass first order.

2 `low_pass_2nd_order`
Low-pass second order.

3 `resonator_frequency`
Resonator frequency.

4 `internal_pll`
Internal PLL.

5 `vco`
Voltage-controlled oscillator (VCO).

6 `resonator_amplitude`
Resonator amplitude.

impulse

//impulse

Properties:	Read
Type:	ZIAdvisorWave
Unit:	None

Reserved for future use - not yet supported.

index

//index

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The 0-based index of the PID on the instrument to use for parameter detection.

pid

//pid/autobw

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled, adjust the demodulator bandwidth to fit best to the specified target bandwidth of the full system. In this case, demod/timeconstant is ignored.

//pid/autolimit

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled, set the instrument PID limits based upon the calculated bw value.

//pid/d

Properties:	Read, Write
Type:	Double
Unit:	(Output Unit . s) / Input Unit

The initial value to use in the Advisor for the differential gain. After optimization has finished it contains the optimal value calculated by the Advisor.

//pid/dlimittimeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

The initial value to use in the Advisor for the differential filter timeconstant gain. After optimization has finished it contains the optimal value calculated by the Advisor.

//pid/i

Properties:	Read, Write
Type:	Double
Unit:	Output Unit / (Input Unit . s)

The initial value to use in the Advisor for the integral gain. After optimization has finished it contains the optimal value calculated by the Advisor.

//pid/mode

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Select PID Advisor mode. Bit encoded: bit 0 — optimize P gain; bit 1 — optimize I gain; bit 2 — optimize D gain; bit 3 — optimize D filter limit

//pid/p

Properties:	Read, Write
Type:	Double
Unit:	Output Unit / Input Unit

The initial value to use in the Advisor for the proportional gain. After optimization has finished it contains the optimal value calculated by the Advisor.

/pid/rate

Properties:	Read, Write
Type:	Double
Unit:	Hz

PID Advisor sampling rate of the PID control loop.

/pid/targetbw

Properties:	Read, Write
Type:	Double
Unit:	Hz

PID system target bandwidth.

/pid/type

Properties:	Read, Write
Type:	String
Unit:	None

HF2 instruments only. Specify whether to model the instrument's PLL or PID hardware unit when dut/source=4 (internal PLL).

pm

/pm

Properties:	Read
Type:	Double
Unit:	deg

Simulated phase margin of the PID with the current settings. The phase margin should be greater than 45 deg and preferably greater than 65 deg for stable conditions.

pmfreq

/pmfreq

Properties:	Read
Type:	Double

Unit: Hz

Simulated phase margin frequency.

progress

//progress

Properties: Read
Type: Double
Unit: None

Reports the progress of a PID Advisor action as a value between 0 and 1.

response

//response

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Set to 1 to calculate the Bode and the step response plot data from the current pid/* parameters (only relevant when auto=0). The module sets response back to 0 when the plot data has been calculated.

stable

//stable

Properties: Read
Type: Integer (64 bit)
Unit: None

If equal to 1, the PID Advisor found a stable solution with the given settings. If equal to 0, the solution was deemed instable - revise your settings and rerun the PID Advisor.

step

//step

Properties: Read
Type: ZIAdvisorWave
Unit: None

The resulting step response data of the PID Advisor's simulation.

targetfail

//targetfail

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

A value of 1 indicates the simulated PID BW is smaller than the Target BW.

tf

//tf/closedloop

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Switch the response calculation mode between closed or open loop.

//tf/input

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Start point for the plant response simulation for open or closed loops.

//tf/output

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

End point for the plant response simulation for open or closed loops.

todevice

//todevice

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

Set to 1 to transfer the calculated PID advisor data to the device, the module will immediately reset the parameter to 0 and configure the instrument's nodes.

tune

//tune

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

If enabled, optimize the instrument's PID parameters so that the noise of the closed- loop system gets minimized. The HF2 doesn't support tuning.

tuner

//tuner/averagetime

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time for a tuner iteration.

//tuner/mode

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Select tuner mode. Bit encoded: bit 0 — tune P gain; bit 1 — tune I gain; bit 2 — tune D gain; bit 3 — tune D filter limit

3.9. Precompensation Advisor Module

The Precompensation Advisor Module provides the functionality available in the LabOne User Interface's Precompensation Tab. In essence the precompensation allows a pre-distortion or pre-emphasis to be applied to a signal before it leaves the instrument, to compensate for undesired distortions caused by the device under test (DUT). The Precompensation Advisor module simulates the precompensation filters in the device, allowing the user to experiment with different filter settings and filter combinations to obtain an optimal output signal, before using the setup in the actual device.

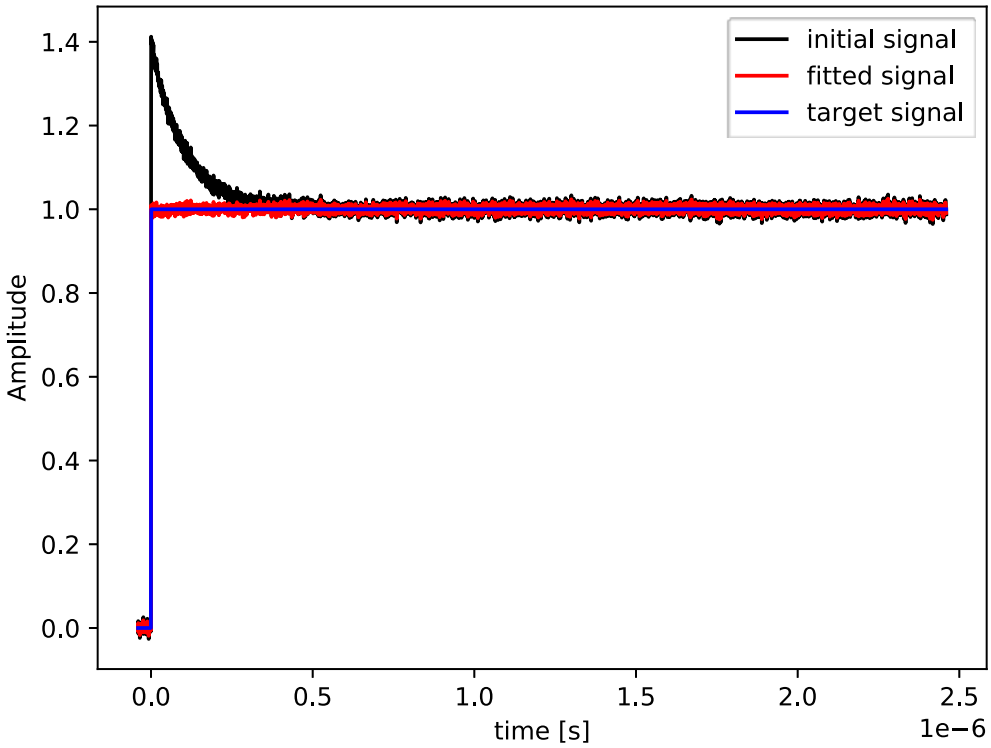


Figure 3.13. The plot was generated by `example_precompensation_curve_fit.py`, an example distributed on our public GitHub repository (https://github.com/zhinst/labone-api-examples).

3.9.1. Precompensation Advisor Module Work-Flow

Precompensation Advisor usage via the LabOne APIs closely follows the work-flow used in the LabOne User Interface.

1. Create an instance of the Precompensation Advisor module (one instance is required for each AWG waveform output in use).
2. Decide which filters to use.
3. Set the coefficients/time constants of the filters.
4. Read and analyze the results of the simulation via the `wave/output`, `wave/output/forwardwave` and `wave/output/backwardwave` parameters.
5. Adjust filter coefficients and repeat the previous two steps until an optimal output waveform is achieved.

Refer to the appropriate user manual for a comprehensive description of the Precompensation Advisor.

Note that with the Precompensation Advisor module, the execute(), finish(), finished() read(), progress(), subscribe() and unsubscribe() commands serve no purpose. Indeed some APIs do not provide all of these commands. Each time one or more filter parameters are changed, the module re-runs the simulation and the results can be read via the `wave/output`, `wave/output/forwardwave` and `wave/output/backwardwave` parameters.

3.9.2. Precompensation Advisor Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the precompensation advisor module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

bounces

//bounces/0/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the bounce compensation filter.

//bounces/0/delay

Properties:	Read, Write
Type:	Double
Unit:	None

Delay between original signal and the bounce echo.

//bounces/0/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the bounce compensation filter.

device

//device

Properties:	Read, Write

Type: String
Unit: None

Device string defining the device on which the compensation is performed.

exponentials

//exponentials/0/amplitude

Properties: Read, Write
Type: Double
Unit: None

Amplitude of the exponential filter.

//exponentials/0/enable

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable the exponential filter.

//exponentials/0/timeconstant

Properties: Read, Write
Type: Double
Unit: Seconds

Time constant (tau) of the exponential filter.

//exponentials/1/amplitude

Properties: Read, Write
Type: Double
Unit: None

Amplitude of the exponential filter.

//exponentials/1/enable

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable the exponential filter.

//exponentials/1/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

//exponentials/2/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

//exponentials/2/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

//exponentials/2/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

//exponentials/3/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

//exponentials/3/enable

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

/exponentials/3/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

/exponentials/4/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

/exponentials/4/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

/exponentials/4/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

/exponentials/5/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

/exponentials/5/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

/exponentials/5/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

/exponentials/6/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

/exponentials/6/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

/exponentials/6/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

/exponentials/7/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

//exponentials/7/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the exponential filter.

//exponentials/7/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the exponential filter.

fir

//fir/coefficients

Properties:	Read, Write
Type:	ZIVectorData
Unit:	None

Vector of FIR filter coefficients. Maximum length 40 elements. The first 8 coefficients are applied to 8 individual samples, whereas the following 32 Coefficients are applied to two consecutive samples each.

//fir/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the FIR filter.

highpass

//highpass/0/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable the high-pass compensation filter.

//highpass/0/timeconstant

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Time constant (tau) of the high-pass compensation filter.

latency

//latency/enable

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable latency simulation for the calculated waves.

//latency/value

Properties:	Read
Type:	Double
Unit:	None

Total delay of the output signal accumulated by all filter stages (read-only).

samplingfreq

//samplingfreq

Properties:	Read
Type:	Double
Unit:	Hz

Sampling frequency for the simulation (read-only). The value comes from the /device/system/clocks/sampleclock/freq node if available. Default is 2.4 GHz.

wave

//wave/input/awgindex

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

Defines with which AWG output the module is associated. This is used for loading an AWG wave as the source.

//wave/input/delay

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Artificial time delay of the simulation input.

//wave/input/gain

Properties:	Read, Write
Type:	Double
Unit:	None

Artificial gain with which to scale the samples of the simulation input.

//wave/input/inputvector

Properties:	Read, Write
Type:	ZIVectorData
Unit:	None

Node to upload a vector of amplitude data used as a signal source. It is assumed the data are equidistantly spaced in time with the sampling rate as defined in the "samplingfreq" node.

//wave/input/length

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Number of points in the simulated wave.

//wave/input/offset

Properties:	Read, Write
Type:	Double
Unit:	V

Artificial vertical offset added to the simulation input.

//wave/input/samplingfreq

Properties:	Read, Write
Type:	Double
Unit:	Hz

The sampling frequency determined by the timestamps from the CSV file.

//wave/input/source

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Type of wave used for the simulation.

0 **`step`**
Step function

1 **`impulse`**
Pulse

2 **`nodes`**
Load AWG with the wave specified by the **`waveindex`** and **`awgindex`** nodes.

3 **`manual`**
Manually loaded wave through the /inputvector node.

//wave/input/statusstring

Properties:	Read
Type:	String
Unit:	None

The status of loading the CSV file.

//wave/input/waveindex

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Determines which AWG wave is loaded from the the AWG output. Internally, all AWG sequencer waves are indexed and stored. With this specifier, the respective AWG wave is loaded into the Simulation.

//wave/output/backwardwave

Properties:	Read
Type:	ZIAdvisorWave
Unit:	None

Initial wave upon which the filters have been applied in the reverse direction. This wave is a simulation of signal path response which can be compensated with the filter settings specified in the respective nodes.

//wave/output/forwardwave

Properties:	Read
Type:	ZIAdvisorWave
Unit:	None

Initial wave upon which the filters have been applied. This wave is a representation of the AWG output when precompensation is enabled with the filter settings specified in the respective nodes.

//wave/output/initialwave

Properties:	Read
Type:	ZIAdvisorWave
Unit:	None

Wave onto which the filters are applied.

3.10. Quantum Analyzer Module

The Quantum Analyzer (QA) module corresponds to the Quantum Analyzer Result Logger tab of LabOne user interface (UI). This nodule allows the user to record multiple measurement shots in its history tab and to apply matrix transformations on the measured complex signals, i.e. I and Q components obtained by weighted integration. It applies transform operations on the measured signals with the following order.

1. Shift or translation
2. Rotation
3. Scaling or dilation

The transform parameters are set by the module nodes and displayed in the control sub-tab of the QA tab in LabOne UI. The equivalent transformed outcome is obtained by matrix multiplication of the corresponding operators as shown in Equation 1.

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{bmatrix} S_I & 0 \\ 0 & S_Q \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{pmatrix} I - I_0 \\ Q - Q_0 \end{pmatrix} \quad (1)$$

where I_0 and Q_0 are shift parameters, θ is rotation angle in degree, and S_I and S_Q are scaling factors. All the measurement shots are recorded in the History sub-tab of the QA tab and can be saved in CSV, HDF5, and MATLAB formats.

3.10.1. Quantum Analyzer Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the quantum analyzer module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

clearhistory

//clearhistory

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Remove all records from the history list.

historylength

//historylength

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Maximum number of entries stored in the measurement history.

rotation

//rotation

Properties:	Read, Write
Type:	Double
Unit:	None

Rotation angle applied to the recorded complex values.

save

//save/csvlocale

Properties:	Read, Write
Type:	String
Unit:	None

The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C": Dot for the decimal point and no digit grouping (default); " (empty string): Use the symbols set in the language and region settings of the computer.

//save/csvseparator

Properties:	Read, Write
Type:	String
Unit:	None

The character to use as CSV separator when saving files in this format.

//save/directory

Properties:	Read, Write
Type:	String
Unit:	None

The base directory where files are saved.

//save/fileformat

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

The format of the file for saving data.

0	**mat** MATLAB
1	**csv** CSV
2	**zview** ZView (Impedance data only)
3	**sxm** SXM (Image format)
4	**hdf5** HDF5

//save/filename

Properties:	Read, Write
Type:	String
Unit:	None

Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g. daq_000.

//save/save

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Initiate the saving of data to file. The saving is done in the background. When the save has finished, the module resets this parameter to 0.

//save/saveonread

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Automatically save the data to file immediately before reading out the data from the module using the read() command. Set this parameter to 1 if you want to save data to file when running the module continuously and performing intermediate reads.

scalingi

//scalingi

Properties:	Read, Write

Type:	Double
Unit:	None

Scaling factor applied to the I component of the recorded data points.

scalingq

/scalingq

Properties:	Read, Write
Type:	Double
Unit:	None

Scaling factor applied to the Q component of the recorded data points.

shifti

/shifti

Properties:	Read, Write
Type:	Double
Unit:	None

Translation shift applied to the I component of the recorded data points.

shiftq

/shiftq

Properties:	Read, Write
Type:	Double
Unit:	None

Translation shift applied to the Q component of the recorded data points.

3.11. Scope Module

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

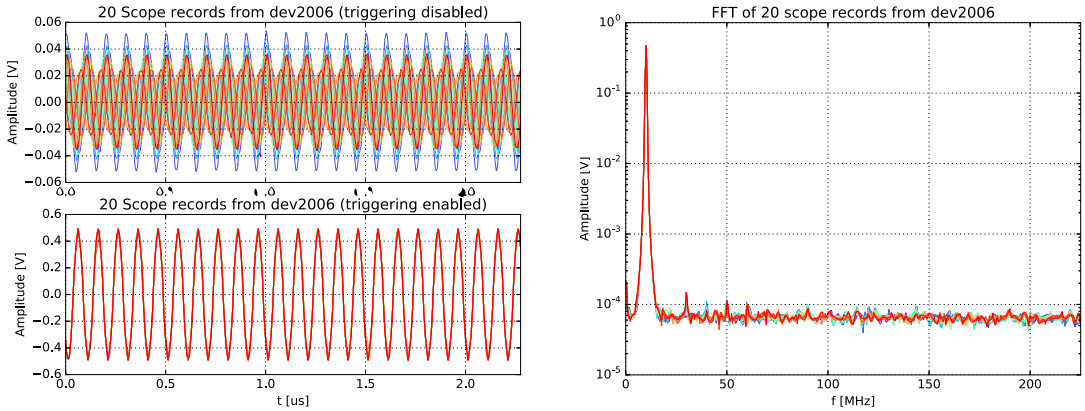


Figure 3.14. The plot was generated by `example_scope.py`, an example distributed on our public GitHub repository (https://github.com/zhinst/labone-api-examples). The example runs the Scope Module in both time and frequency mode with scope record averaging enabled.

3.11.1. Introduction to Scope Data Transfer

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

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

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

Segmented Mode

Additionally, MF and UHF instruments which have the DIG option enabled can optionally record data in "segmented" mode which allows back-to-back measurements with very small hold-off times between measurements. Segmented mode enables the user to make a fixed number of measurements (the segments), which are stored completely in the memory of the instrument, before they are transferred to the Data Server (this overcomes the data transfer rate limitation of

the device's connected physical interface, e.g., USB or 1GbE for a fixed number of measurements). The advantage to this mode is precisely that the hold-off time, i.e. the delay between two measurements, can be very low. Data recorded in segmented mode is still available from the instrument node /DEV..../SCOPES/n/WAVE as for non-segmented data, but requires an additional reshaping described in Section 3.11.6.

Scope Data Nomenclature

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

Wave — The name of the leaf (/DEV..../SCOPES/n/WAVE) in the device node tree that contains scope data pushed from the instrument's firmware to the Data Server. The data structure returned by this node is defined by the API Level of the connected session. It is also the name of the structure member in scope data structures that holds the actual scope data samples. See Scope Data Structures below for detailed information.

Record — Refers to one complete scope data element returned by the Scope Module. It may consist of one or multiple segments.

Segment — A segment is a completely assembled and scaled wave. If the instrument's scope is used in segmented mode, each record will consist of multiple segments. If not used in segmented mode, each record comprises of a single segment and the terms record and segment can be used interchangeably.

Block — When the length of data (/DEV..../SCOPES/n/LENGTH) in a scope segment is very large the segment returned by the device node (/DEV..../SCOPES/n/WAVE) is split into multiple blocks. When using the poll/subscribe method the user must assemble these blocks; the Scope Module assembles them for the user into complete segments.

Shot — The term shot is often used when discussing data acquired from laboratory oscilloscopes, we try to avoid it in the following in order to more easily distinguish between records and segments when recording in segmented mode.

Scope Data Structures

The device node /DEV..../SCOPES/n/WAVE (and /DEV..../SCOPES/n/STREAM/SAMPLE DIG Option enabled, API level > 4) return the following data structures[2] based on the API level used by the session:

ScopeWave — API Level 1, HF2 only. The simplest scope data structure. The **`wave`** structure member in the data structure always has a fixed length of 2048. Scope records are never split into multiple blocks; no assembly required. The HF2 does not support segmented recording; a segment is equivalent to a record.

ZIScopeWave — API Level 4. An extended scope data structure used with MF and UHF instruments. The data in the **`wave`** structure member consists of one scope block; for long scope segments, complete scope segments must be assembled by combining these blocks. The data in **`wave`** is not scaled or offset.

ZIScopeWaveEx — API Level > 5. As for ZIScopeWave, but contains the additional structure member **`channelOffset`**.

The Scope Module always returns scope data in the ZIScopeWaveEx[2] format, regardless of which supported API level (1, >4) was used in the session where the Scope Module was instantiated.

[2]Please refere to the Labone API documentation

However, the data in the `wave` structure member always consists of complete segments (it does not need to be reassembled from multiple blocks). More differences between the data returned by the node /DEV..../SCOPES/n/WAVE and the Scope Module are highlighted in Section 3.11.2.

3.11.2. Advantages of the Scope Module

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

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

3.11.3. Working with Scope Module

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

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

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

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

4. Configure the instrument's scope as required for the measurement. If recording signals other than hardware channel signals (such as a PID's error or a demodulator R output), be sure to configure the /DEV..../SCOPES/n/CHANNELS/n/LIMIT\{LOWER UPPER} accordingly to obtain the best bit resolution in the scope).
5. Configure the ***** parameters as required, in particular:
   - Set **mode** in order to specify whether to return time or frequency domain data. See Section 3.11.4 for more information on the Scope Module's modes.
   - Set the **historylength** parameter to tell the Scope Module to only return a certain number of records. Note, as the Scope Module is acquiring data the **records** output parameter may grow larger than **historylength**; the Scope Module will return the last number of records acquired.
   - Set **averager/weight** to a value larger than 1 to enable averaging of scope records, see Section 3.11.5.
6. Enable the scope (if not previously enabled) and call Scope Module **execute()** to start acquiring data.
7. Wait for the Scope Module to acquire the specified number of records. Note, if certain scope parameters are modified during recording, the history of the Scope Module will be cleared, Section 3.11.7 for the list of parameters that trigger a Scope Module reset.
8. Call Scope Module **read()** to transfer data from the Module to the client. Data may be read out using **read()** before acquisition is complete (advanced use). Note, an intermediate read will create a copy in the client of the incomplete record which could be critical for memory consumption in the case of very long scope lengths or high segment counts. The data structure returned by **read()** is of type ZIScopeWaveEx[2]
9. Check the flags of each record indicating whether any problems occurred during acquisition.
10. Extract the data for each recorded scope channel and, if recording data in segmented mode, reshape the wave data to allow easier access to multi-segment records (see Section 3.11.6) Note, the scope data structure only returns data for enabled scope channels.

Data Acquisition and Transfer Speed

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

3.11.4. Scope Module Modes

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

The Scope Module does not return an array consisting of the points in time (time mode) or frequencies (FFT mode) corresponding to the samples in ZIScopeWaveEx[2]. These can be constructed as arrays of n points, where n is the configured scope length (/DEV.../SCOPES/n/LENGTH)) spanning the intervals:

Time mode [0, **dt*totalsamples**], where **dt** and **totalsamples** are fields in ZIScopeWaveEx[2].
In order to get a time array relative to the trigger position, **(timestamp - triggertimestamp)/float(clockbase)** must be subtracted from the

times in the interval, where **timestamp** and **triggerstamp** are fields in ZIScopeWaveEx[2].

FFT mode [0, **(clockbase/2^scope_time)/2**], where **scope_time** is the value of the device node /DEV..../SCOPES/n/TIME. and **clockbase** is the value of /DEV..../CLOCKBASE.

3.11.5. Averaging

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

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

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

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

The average calculation can be restarted by setting **averager/restart** to 1. It is currently not possible to tell how many scope segments have been averaged (since the reset). In general, however, the way to track the current scope record is via the **sequenceNumber** field in ZIScopeWaveEx[2].

3.11.6. Segmented Recording

When the instrument's scope runs segmented mode the **wave** structure member in the ZIScopeWaveEx[2] is an array consisting of **length*segment_count** where **length** is the value of /DEV..../SCOPES/0/LENGTH and **num_segments** is /DEV..../SCOPES/n/SEGMENTS/COUNT. This is equal to the value of the **totalsamples** structure member.

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

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

3.11.7. Scope Parameters that reset the Scope Module

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

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

3.11.8. Device-specific considerations

Scope Module Use on HF2 Instruments

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

Scope Module Use on MF and UHF Instruments

For MF and UHF instruments no special considerations must be made except that LabOne API Levels 4 is not supported by the Scope Module; a higher API level must be used.

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

3.11.9. Scope Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the scope module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

averager

//averager/resamplingmode

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Specifies the resampling mode. When averaging scope data recorded at a low sampling rate that is aligned by a high resolution trigger, scope data must be resampled to keep the corresponding samples between averaged recordings aligned correctly in time relative to the trigger time.

0 `linear`
Linear interpolation

1 `pchip`
PCHIP interpolation

//averager/restart

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

Set to 1 to reset the averager. The module sets averager/restart back to 0 automatically.

/averager/weight

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Specify the averaging behaviour. weight=0: Averaging disabled. weight>1: Moving average, updating last history entry.

clearhistory

/clearhistory

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Remove all records from the history list.

error

/error

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

Indicates whether an error occurred whilst processing the current scope record; set to non-zero when a scope flag indicates an error. The value indicates the accumulated error for all the processed segments in the current record and is reset for every new incoming scope record. It corresponds to the status LED in the LabOne User Interface's Scope tab - API users are recommended to use the flags structure member in ZIScopeWaveEx instead of this output parameter.

externalscaling

/externalscaling

Properties:	Read, Write

Type: Double
Unit: None

Scaling to apply to the scope data transferred over API level 1 connection. Only relevant for HF2 Instruments.

fft

/fft/power

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable calculation of the power value.

/fft/spectraldensity

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Enable calculation of the spectral density value.

/fft/window

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

FFT Window

0 `rectangular`
Rectangular
1 `hann`
Hann (default)
2 `hamming`
Hamming
3 `blackman_harris`
Blackman Harris
16 `exponential`
Exponential (ring-down)
17 `cos`
Cosine (ring-down)
18 `cos_squared`

Cosine squared (ring-down)

historylength

//historylength

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Maximum number of entries stored in the measurement history.

lastreplace

//lastreplace

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Reserved for LabOne User Interface use.

mode

//mode

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

The Scope Module's data processing mode.

0 `passthrough`

Passthrough: scope segments assembled and returned unprocessed, non-interleaved.

1 `exp_moving_average`

Moving average: entire scope recording assembled, scaling applied, averager if enabled (see averager/weight), data returned in float non-interleaved format.

2 Reserved for future use (average n segments).

3 `fft`

FFT, same as mode 1, except an FFT is applied to every segment of the scope recording before averaging. See the fft/* parameters for FFT parameters.

records

//records

Properties:	Read
Type:	Integer (64 bit)
Unit:	None

The number of scope records that have been processed by the Scope Module since execute() was called or a critical scope setting has been modified (see manual for a list of scope settings that trigger a reset).

save

//save/csvlocale

Properties:	Read, Write
Type:	String
Unit:	None

The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C": Dot for the decimal point and no digit grouping (default); " (empty string): Use the symbols set in the language and region settings of the computer.

//save/csvseparator

Properties:	Read, Write
Type:	String
Unit:	None

The character to use as CSV separator when saving files in this format.

//save/directory

Properties:	Read, Write
Type:	String
Unit:	None

The base directory where files are saved.

//save/fileformat

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

The format of the file for saving data.

0	**mat** MATLAB
1	**csv** CSV
2	**zview** ZView (Impedance data only)
3	**sxm** SXM (Image format)
4	**hdf5** HDF5

//save/filename

Properties:	Read, Write
Type:	String
Unit:	None

Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g. daq_000.

//save/save

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Initiate the saving of data to file. The saving is done in the background. When the save has finished, the module resets this parameter to 0.

//save/saveonread

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Automatically save the data to file immediately before reading out the data from the module using the read() command. Set this parameter to 1 if you want to save data to file when running the module continuously and performing intermediate reads.

3.12. Sweeper Module

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

3.12.1. Configuring the Sweeper

In this section we briefly describe how to configure the Sweeper Module. See Section 3.12.2 for a full list of the Sweeper's parameters and description of the Sweeper's outputs.

Specifying the Instrument Setting to Sweep

The Sweeper's `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 `samplecount` times within the **range** of values specified by `start` and `stop`. The `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 `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. 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. 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. 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 `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 `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, **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 **settling/** branch: **settling/time** and **settling/inaccuracy**.

The **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 **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 **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 **settling/inaccuracy** and the demodulator filter order, the number of demodulator filter time constants to wait is calculated and written to **settling/tc** (upon calling the module's **execute()** command) which can then be read back by the user. See Section 3.12.2 for recommended values of **settling/inaccuracy**. The relationship between **settling/inaccuracy** and **settling/tc** is plotted in Figure 3.15.

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 1. For a frequency sweep, the **settling/inaccuracy** parameter will tend to influence the settling time at lower frequencies, whereas **settling/time** will tend to influence the settling time at higher frequencies.

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 **settling/tc** and **settling/time** (see Equation 1).

$$t_s = \max\{\text{tc} \times (\text{settling/tc}), (\text{settling/time})\} \qquad (2)$$

Note

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

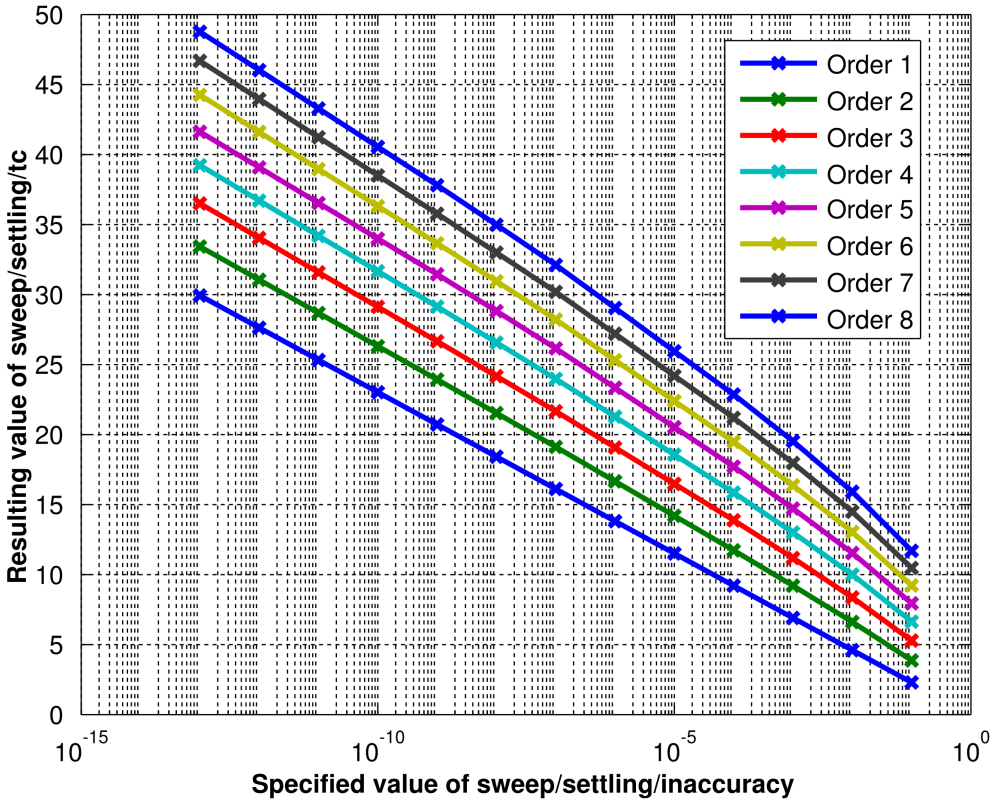


Figure 3.15. A plot showing the values of the Sweeper's `settling/tc` as calculated from `settling/inaccuracy` parameter and their dependency on demodulator order filter.

Specifying which Data to Measure

Which measurement data is actually returned by the Sweeper's **`read`** command is configured by subscribing to node path 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 **`averaging/`** branch.

The **`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 **`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.

The number of samples N used to average one sweeper measurement point is determined by the parameters **`averaging/time`**, **`averaging/tc`**, and **`averaging/sample`** as well as the **`rate`** of data transfer from the instrument to the data server (see Equation 2).

$$N = \max\{\mathrm{tc} \times (\mathrm{averaging/tc}) \times \mathrm{rate}, (\mathrm{averaging/time}) \times \mathrm{rate}, (\mathrm{averaging/sample})\} \quad (3)$$

Note

Note, the value of the demodulator filter's time constant may be controlled by the Sweeper depending on the value of **bandwidthcontrol** and **bandwidth**, see the section above called the section called "Controlling how the Sweeper sets the Demodulator's Time Constant ". For a frequency sweep, the **averaging/tc** parameter will tend to influence the number of samples recorded at lower frequencies, whereas **averaging/sample** will influence averaging behavior at higher frequencies.

An Explanation of Settling and Averaging Times in a Frequency Sweep

Figure 3.16 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:

- **gridnode** is set to an oscillator frequency, e.g., **/dev123/oscs/0/freq**.
- **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.
- **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 **settling/tc** (this is calculated by the Sweeper Module from the specified **settling/inaccuracy** parameter). The purple bars correspond to the time specified by the **settling/time** parameter. The sweeper will wait for the maximum of these two times according to Equation 1. When measuring at lower frequencies the Sweeper sets a smaller demodulator filter bandwidth (due to automatic **bandwidthcontrol**) corresponding to a larger demodulator filter time constant. Therefore, the **settling/tc** parameter dominates the settling time used by the Sweeper at low frequencies and at high frequencies the **settling/time** parameter takes effect. Note, that the light blue bars corresponding to the value of **settling/tc** get shorter for each measurement point (larger frequency used → shorter time constant required), whereas the purple bars corresponding to **settling/time** stay a constant length for each measurement point. Similarly, the **averaging/tc** parameter (yellow bars) dominates the Sweeper's averaging behavior at low frequencies, whereas **averaging/samples** (green bars) specifies the behavior at higher frequencies, see also Equation 2.

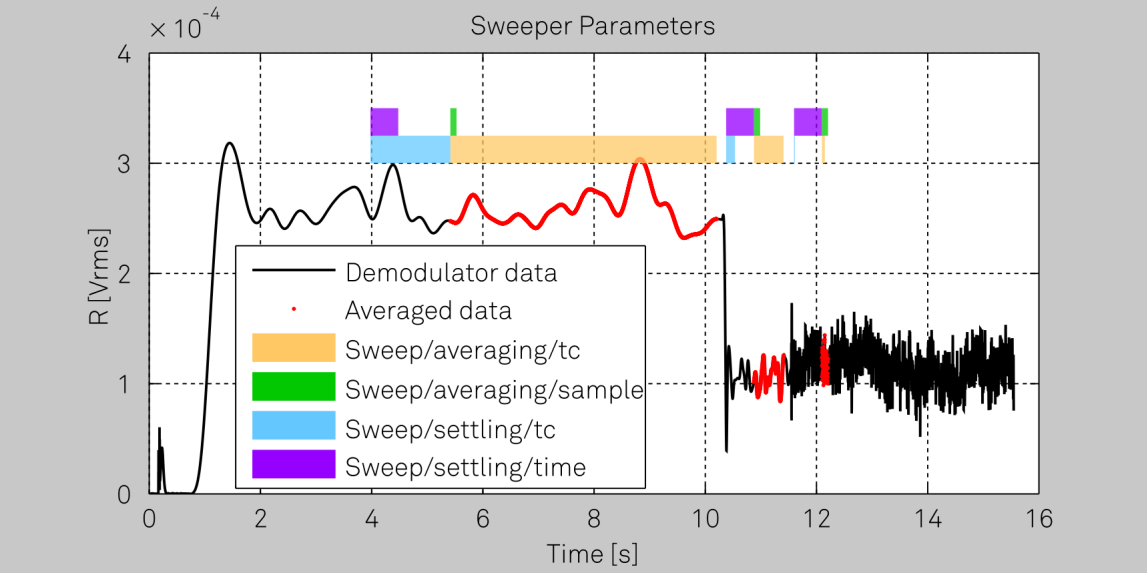


Figure 3.16. Plot demonstrating how the Sweeper records three measurement points from demodulator data when using automatic bandwidth control in a frequency sweep. Please see the section called "An Explanation of Settling and Averaging Times in a Frequency Sweep", for a detailed explanation.

Average Power and Standard Deviation of the Measured Data

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

3.12.2. Sweeper Module Node Tree

The following section contains reference documentation for the settings and measurement data available on the sweeper module.

Since these settings and data streams may be written and read using the LabOne APIs (Application Programming Interfaces) this section is of particular interest to users who would like to perform measurements programmatically via LabVIEW, Python, MATLAB, .NET or C.

averaging

//averaging/sample

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	Samples

Sets the number of data samples per sweeper parameter point that is considered in the measurement.

/averaging/tc

Properties:	Read, Write
Type:	Double
Unit:	TC

Sets the effective number of time constants per sweeper parameter point that is considered in the measurement.

/averaging/time

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Sets the effective measurement time per sweeper parameter point that is considered in the measurement.

awgcontrol

/awgcontrol

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable AWG control for sweeper. If enabled the sweeper will automatically start the AWG and records the sweep sample based on the even index in hwtrigger.

bandwidth

/bandwidth

Properties:	Read, Write
Type:	Double
Unit:	Hz

Defines the measurement bandwidth when using Fixed bandwidth mode (sweep/bandwidthcontrol=1), and corresponds to the noise equivalent power bandwidth (NEP).

bandwidthcontrol

/bandwidthcontrol

Properties:	Read, Write

Type: Integer (enumerated)
Unit: None

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`
Manual (the sweeper module leaves the demodulator bandwidth settings entirely untouched)

1 `fixed`
Fixed (use the value from sweep/bandwidth)

2 `auto`
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).

bandwidthoverlap

//bandwidthoverlap

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

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.

clearhistory

//clearhistory

Properties: Read, Write
Type: Integer (64 bit)
Unit: None

Remove all records from the history list.

device

//device

Properties: Read, Write
Type: String
Unit: None

The device ID to perform the sweep on, e.g., dev123 (compulsory parameter, this parameter must be set first).

endless

//endless

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable Endless mode; run the sweeper continuously.

filtermode

//filtermode

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Selects the filter mode.

0	`application` Application (the sweeper sets the filters and other parameters automatically)
1	`advanced` Advanced (the sweeper uses manually configured parameters)

gridnode

//gridnode

Properties:	Read, Write
Type:	String
Unit:	Node

The device parameter (specified by node) to be swept, e.g., "oscs/0/freq".

historylength

//historylength

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

Maximum number of entries stored in the measurement history.

loopcount

//loopcount

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The number of sweeps to perform.

maxbandwidth

//maxbandwidth

Properties:	Read, Write
Type:	Double
Unit:	Hz

Specifies the maximum bandwidth used when in Auto bandwidth mode (sweep/bandwidthcontrol=2). The default is 1.25 MHz.

omegasuppression

//omegasuppression

Properties:	Read, Write
Type:	Double
Unit:	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.

order

//order

Properties:	Read, Write

Type:	Integer (64 bit)
Unit:	None

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).

phaseunwrap

//phaseunwrap

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Enable unwrapping of slowly changing phase evolutions around the +/-180 degree boundary.

remainingtime

//remainingtime

Properties:	Read
Type:	Double
Unit:	Seconds

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.

samplecount

//samplecount

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

The number of measurement points to set the sweep on.

save

//save/csvlocale

Properties:	Read, Write

Type: String
Unit: None

The locale to use for the decimal point character and digit grouping character for numerical values in CSV files: "C": Dot for the decimal point and no digit grouping (default); " (empty string): Use the symbols set in the language and region settings of the computer.

/save/csvseparator

Properties: Read, Write
Type: String
Unit: None

The character to use as CSV separator when saving files in this format.

/save/directory

Properties: Read, Write
Type: String
Unit: None

The base directory where files are saved.

/save/fileformat

Properties: Read, Write
Type: Integer (enumerated)
Unit: None

The format of the file for saving data.

0 `mat`
MATLAB
1 `csv`
CSV
2 `zview`
ZView (Impedance data only)
3 `sxm`
SXM (Image format)
4 `hdf5`
HDF5

/save/filename

Properties: Read, Write

Type:	String
Unit:	None

Defines the sub-directory where files are saved. The actual sub-directory has this name with a sequence count (per save) appended, e.g. daq_000.

/save/save

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Initiate the saving of data to file. The saving is done in the background. When the save has finished, the module resets this parameter to 0.

/save/saveonread

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

Automatically save the data to file immediately before reading out the data from the module using the read() command. Set this parameter to 1 if you want to save data to file when running the module continuously and performing intermediate reads.

scan

/scan

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Selects the scanning type.

0	`sequential` Sequential (incremental scanning from start to stop value)
1	`binary` Binary (Non-sequential sweep continues increase of resolution over entire range)
2	`bidirectional` Bidirectional (Sequential sweep from Start to Stop value and back to Start again)
3	`reverse` Reverse (reverse sequential scanning from stop to start value)

settling

//settling/inaccuracy

Properties:	Read, Write
Type:	Double
Unit:	None

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. See programming manual for the relationship between sweep/settling/inaccuracy and sweep/settling/tc.

//settling/tc

Properties:	Read, Write
Type:	Double
Unit:	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. The maximum between this value and sweep/settling/time is taken as effective settling time.

//settling/time

Properties:	Read, Write
Type:	Double
Unit:	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/tc is taken as effective settling time.

sincfilter

//sincfilter

Properties:	Read, Write
Type:	Integer (64 bit)
Unit:	None

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.

start

/*/start

Properties:	Read, Write
Type:	Double
Unit:	Many

The start value of the sweep parameter.

stop

/*/stop

Properties:	Read, Write
Type:	Double
Unit:	Many

The stop value of the sweep parameter.

xmapping

/*/xmapping

Properties:	Read, Write
Type:	Integer (enumerated)
Unit:	None

Selects the spacing of the grid used by sweep/gridnode (the sweep parameter).

0	`linear` Linear
1	`log` Logarithmic distribution of sweep parameter values

Chapter 4. MATLAB Programming

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

This chapter aims to help you get started using Zurich Instruments LabOne's MATLAB API, **ziDAQ**, to control your instrument.

Please refer to:

- Section 4.1 for help installing the LabOne MATLAB API.
- Section 4.2 for help getting Started with the LabOne MATLAB API and running the examples.
- Section 4.3 for some LabOne MATLAB API tips and tricks.
- Section 4.4 for help troubleshooting the LabOne MATLAB API.

Note

For a full reference of the MATLAB API visit the LabOne API documentation. The LabOne API documentation is available within your LabOne Software or can be accessed online at the Zurich Intruments website (www.zhinst.com)

Note

The MATLAB examples can be downloaded from our GitHub repository.

Note

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

4.1. Installing the LabOne MATLAB API

4.1.1. Requirements

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

1. 32 or 64-bit Windows with MATLAB R2009b or newer.
2. 64-bit Linux with MATLAB R2016b or newer.
3. 64-bit macOS and MATLAB R2013b or newer.

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

Separate MATLAB Package

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

1. Use the MATLAB API 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.
2. Use the MATLAB API on a PC where you do not have administrator rights.

4.1.2. Windows, Linux or Mac

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

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

   If using a LabOne installation on Windows this is typically:

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

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

   ```
   [PATH]/LabOne64/API/MATLAB/
   ```

2. In the MATLAB Command Window, run the MATLAB script **ziAddPath** located in the **MATLAB** directory:

   ```
   >> ziAddPath;
   ```

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

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

```
To make this configuration persistent across MATLAB sessions either:

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

   C:\Program Files\Zurich Instruments\LabOne\API\MATLAB\

or

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

   run('C:\Program Files\Zurich Instruments\LabOne\API\MATLAB\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**):
   1. Run the **pathtool** and click "Add with Subfolders". Browse to the "MATLAB" directory that was located above in Step 1 and click "OK".
   2. Edit your **startup.m** to contain the line indicated in the output from Step 2 above. For more help on MATLAB's **startup.m** file, type the following in MATLAB's Command Window:

      ```
      >> docsearch('startup.m')
      ```
4. Verify your MATLAB configuration as described in Section 4.1.3.

4.1.3. Verifying Successful MATLAB Configuration

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

1. Ensure that the correct Data Server is running for your HDAWG, HF2 or UHF Instrument (the Data Server on MF Instruments starts when the device is powered on). The quickest way to check is to start the User Interface for your device, see Section 1.2 for more details.
2. Proceed either of the following two ways:
   a. The easiest way to verify correct configuration is run one of the MATLAB API's examples (see <<pm.matlab.github_repository). 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 an API session to the Data Server without device communication.

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

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

      Note, using **'localhost'** above assumes that the Data Server is running on the same computer from which you are using MATLAB. See Section 1.4.1 for information about ports and host names when connecting to the Data Server. For MFLI instruments the hostname/IP address of the MFLI instrument must be provided (the value of **mf-**

`hostname`), see Section 1.4.1 and the Getting Started chapter of the MFLI User Manual for more information.

3. If no error is reported then MATLAB is correctly configured to use `ziDAQ` - congratulations! Otherwise, please try the steps listed in Section 4.4.

4.2. Getting Started with the LabOne MATLAB API

This section introduces the user to the LabOne MATLAB API.

4.2.1. Contents of the LabOne MATLAB API

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

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

in the MATLAB Command Window in the LabOne MATLAB API directory or take a look at the LabOne API documentation for a detailed overview.

MATLAB Driver Naming

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

4.2.2. Using the Built-in Documentation

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

```
>> help ziDAQ
```

```
>> doc ziDAQ
```

This documentation is located in the file **MATLAB/Driver/ziDAQ.m**. See the LabOne API documentation for a detailed overview.

4.2.3. Running the Examples

Prerequisites for running the MATLAB examples:

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

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

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

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

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

Most examples will also plot some data in a MATLAB figure, see Figure 4.1 for an example. If you encounter an error message please ensure that the Section 4.2.3 are fulfilled and see Section 4.4 for help troubleshooting the error.

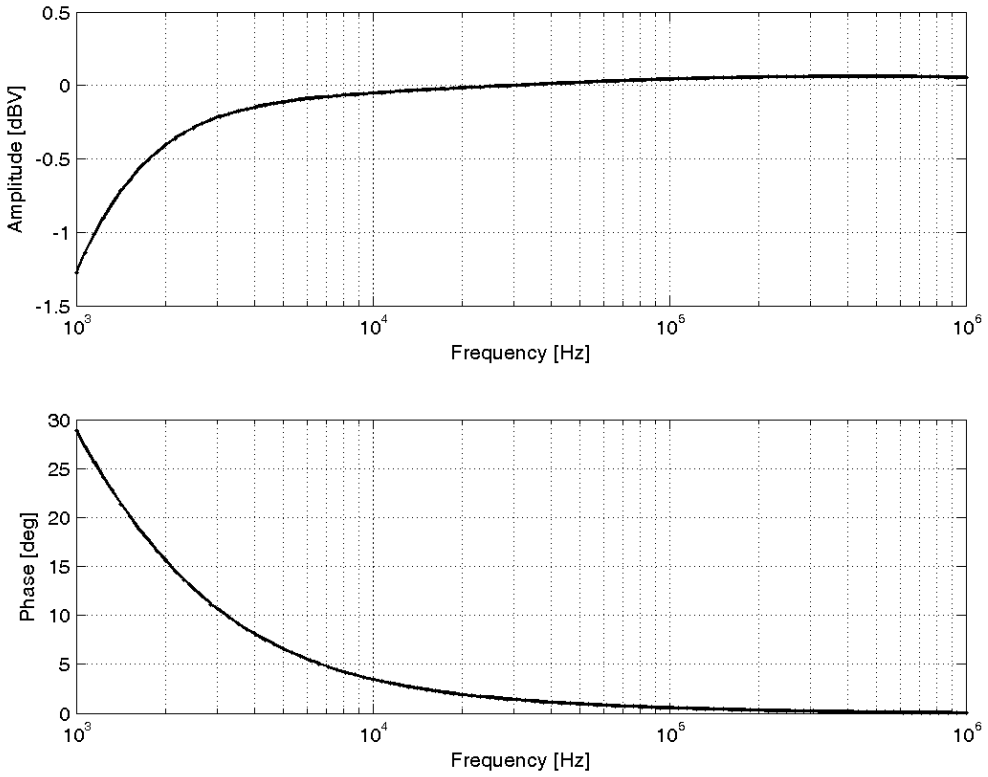


Figure 4.1. The plot produced by the LabOne MATLAB API example `example_sweeper.m`; the plots show the instruments demodulator output when performing a frequency sweep over a simple feedback cable.

Note

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

4.2.4. Using ziCore Modules in the LabOne MATLAB API

In the LabOne MATLAB API **ziCore** Modules are configured and controlled via MATLAB "handles". For example, in order to use the Sweeper Module a handle is created via:

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

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

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

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

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

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

Note

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

4.2.5. Enabling Logging in the LabOne MATLAB API

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

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

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

```
trace:0, debug:1, info:2, status:3, warning:4, error:5, fatal: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 are located in the directory **C:\Users\[USER]\AppData\Local\Temp\Zurich Instruments\LabOne** Note that **AppData** is a hidden directory. The easiest way to find it is to open a File Explorer window and type the text **%AppData%\..** in the address bar, and navigate from there. The directory contains folders containing log files from various LabOne components, in particular, the **ziDAQLog** folder contains logs from the LabOne MATLAB API. On Linux, the logs can be found at "/tmp/ziDAQLog_USERNAME", where "USERNAME" is the same as the output of the "whoami" command.

4.3. LabOne MATLAB API Tips and Tricks

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

The structure of ziDAQ commands.

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

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

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

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

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

See the LabOne API documentation for a detailed overview.

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...
```

4.4. Troubleshooting the LabOne MATLAB API

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

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

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

```
>> path
```

and repeating the steps to configure MATLAB's search path in Section 4.1.2.

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

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

```
>> path
```

and repeating the steps to configure MATLAB's search path in Section 4.1.2.

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

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

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

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

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

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

where **hostname** should be replaced by the IP of the computer where the Data Server is running, **port** is specified as in Section 1.4.1. and **api_level** is specified as described in Section 1.4.2.

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

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

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

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

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

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

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

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

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

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

Chapter 5. Python Programming

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

This chapter aims to help you get started using the Zurich Instruments LabOne Python API to control your instrument.

Please refer to:

- Section 5.1 for help installing the LabOne Python API.
- Section 5.2 for help getting started with the LabOne Python API and running the examples.
- Section 5.3 for LabOne Python API tips and tricks.

LabOne API documentation

For a full reference of the Python API visit the LabOne API documentation. The LabOne API documentation is available within your LabOne Software or can be accessed online at the Zurich Intruments website (www.zhinst.com).

LabOne API Examples

To see the Python API in action take a look at some of the examples that we published on our public GitHub repository (https://github.com/zhinst/labone-api-examples).

About Python

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

This chapter and the provided examples are not intended to be a Python tutorial. For help getting started with Python itself, see either the Python Tutorial or one of the many online resources, for example, the learnpython.org.

5.1. Installing the LabOne Python API

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

```
pip install zhinst
```

5.1.1. Requirements

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

1. One of the following supported platforms and Python versions:
   1. Windows 10, **x86_64**, with a Python 3.5-3.9 installation.
   2. GNU/Linux with glibc 2.17 or newer (CentOS/RHEL 7+, all recent versions of Debian and Ubuntu). **x86_64** wheels are available for Python 3.5-3.9. **aarch64** wheels are available for Python 3.7-3.9.
   3. macOS 10.11+ **x86_64** and Python 3.5-3.9. macOS 11+ **arm64** (Apple Silicon) wheels are provided for Python 3.9+.
2. Installation on Linux requires **pip** 19.3+ for support of the **manylinux2014** platform tag. In case of problems, please try to install the package in a virtual environment with latest **pip**:

   ```
   $ python3 -m venv venv
   $ . venv/bin/activate
   $ pip install --upgrade pip
   $ pip install zhinst
   ```
3. The **numpy** Python package.

5.1.2. Recommended Python Packages

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

1. **matplotlib** - recommended to plot the output from many of the provided examples.
2. **scipy** - recommended to load data saved from the LabOne UI in binary MATLAB format (.mat).

5.1.3. Installation (Windows, Linux, macOS)

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

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

   ```
   import sys
   print(sys.executable)
   ```

   On Windows this will print something similar to:

   ```
   C:\Python37\python.exe
   ```
2. Install the **zhinst** package. Using the full path to the Python executable, PATH_TO_PYTHON_EXE, as determined above in Step 1, open a command prompt and run Python with the **pip** module to install the **zhinst** package:

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

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

Global Installation as Administrator

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

5.1.4. Offline Installation

To install `zhinst` package on a computer without access to the internet, please download the correct wheel file for your system and Python version from https://pypi.org/project/zhinst/ from another computer and copy it to the offline computer. If the `numpy` package is not yet installed, it can be downloaded from https://pypi.org/project/numpy/. Then the wheels can be installed as described above using `pip`, except that the name of the wheel file must be provided as the last argument to `pip` instead of the name of the package, `zhinst`.

5.2. Getting Started with the LabOne Python API

This section introduces the user to the LabOne Python API.

5.2.1. Contents of the LabOne Python API

Alongside the binary extension **zhinst.core** for interfacing with Zurich Instruments Data Servers and devices, the LabOne Python API includes utility functions.

5.2.2. Using the Built-in Documentation

zhinst.core's built-in documentation can be accessed using the **help** command in a python interactive shell:

- On module level:

```
>>> import zhinst.core
>>> help(zhinst.core)
```

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

```
>>> import zhinst.core
>>> help(zhinst.core.SweeperModule)
```

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

```
>>> import zhinst.core
>>> help(zhinst.core.ziDAQServer.poll)
```

See the LabOne API documentation for a full documentation.

5.2.3. Using ziCore Modules in the LabOne Python API

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

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

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

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

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

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

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

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

5.2.4. Enabling Logging in the LabOne Python API

Logging from the API is not enabled by default upon initializing a server session. 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, debug:1, info:2, status:3, warning:4, error:5, fatal:6.
```

It is also possible for the user to write their own messages directly to the LabOne Python API 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 are located in the directory `C:\Users\[USER]\AppData\Local\Temp\Zurich Instruments\LabOne` Note that `AppData` is a hidden directory. The easiest way to find it is to open a File Explorer window and type the text `%AppData%\..` in the address bar, and navigate from there. The directory contains folders containing log files from various LabOne components, in particular, the `ziPythonLog` folder contains logs from the LabOne Python API. On Linux, the logs can be found at "/tmp/ziPythonLog_USERNAME", where "USERNAME" is the same as the output of the "whoami" command.

5.3. LabOne Python API Tips and Tricks

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

Data Structures returned by the LabOne Python API.

The output arguments that the LabOne Python API 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 the LabOne API documentation for reference documentation on these commands.

Chapter 6. LabVIEW Programming

Interfacing with your Zurich Instruments device via National Instruments' LabVIEW® is an efficient choice in terms of development time and run-time performance. LabVIEW is a graphical programming language designed to interface with laboratory equipment via so-called VIs ("virtual instruments"), whose key strength is the ease of displaying dynamic signals obtained from your instrument.

This chapter aims to help you get started using the Zurich Instruments LabOne LabVIEW API to control your instrument.

Please refer to:

- Section 6.1 for help installing the LabOne LabVIEW API.
- Section 6.2 for help getting started with the LabOne LabVIEW API and running the LabOne Example VIs.
- Section 6.3 for some LabVIEW programming tips and tricks.

Note

This section and the provided examples are not intended to be a general LabVIEW tutorial. See, for example, the National Instruments website for help to get started programming with LabVIEW.

6.1. Installing the LabOne LabVIEW API

6.1.1. Requirements

One of the following platforms and LabVIEW versions is required to use the LabOne LabVIEW API:

1. Windows with 32- or 64-bit LabVIEW 2009 or newer.
2. Linux with 64-bit LabVIEW 2010 or newer.
3. macOS with LabVIEW 2010 or newer.

The LabOne LabVIEW API is included in a standard LabOne installation and is also available as a separate package (see below, Separate LabVIEW Package). In order to make the LabOne LabVIEW API available for use within LabVIEW, a directory needs to be copied to a specific directory of your LabVIEW installation. Both the main LabOne installer and the separate LabOne LabVIEW API package are available from Zurich Instruments' download page.

Separate LabVIEW Package

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

1. Use the LabVIEW API on macOS (the main LabOne installer is not available for macOS).
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.

6.1.2. Windows Installation

1. Locate the **instr.lib** directory in your LabVIEW installation and delete any previous Zurich Instruments API directories. The **instr.lib** directory is typically located at:

   ```
   C:\Program Files\National Instruments\LabVIEW 201x\instr.lib\
   ```

   Previous Zurich Instruments installations will be directories located in the **instr.lib** directory that are named either:

   - `Zurich Instruments HF2`, or
   - `Zurich Instruments LabOne`

   These folders may simply be deleted (administrator rights required).

2. On Windows, either navigate to the **API\LabVIEW** subdirectory of your LabOne installation or, in the case of the separate installer (see Separate LabVIEW Package), the directory of the unzipped LabOne LabVIEW package, and copy the subdirectory

   ```
   Zurich Instruments LabOne
   ```

   to the **instr.lib** directory in your LabVIEW installation as located in Step 1. Note, you will need administrator rights to copy to this directory.

   In the case of copying from a LabOne installation, this folder is typically located at:

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

3. Restart LabVIEW and verify your installation as described in Section 6.1.4.

6.1.3. Linux and Mac Installation

1. Locate the `instr.lib` directory in your LabVIEW installation and remove any previous Zurich Instruments API installations. The `instr.lib` directory is typically located on Linux at:

   `/usr/local/natinst/LabVIEW-201x/instr.lib/`

   and on macOS at:

   `/Applications/National Instruments/LabVIEW 201x/instr.lib/`

   Previous Zurich Instruments installations will be folders located in the `instr.lib` directory that are named either:

   - `Zurich Instruments HF2`, or
   - `Zurich Instruments LabOne`.

   These folders may simply be deleted (administrator rights required).

2. Navigate to the path where you unpacked LabOne or the Separate LabVIEW Package and copy the subdirectory

   `Zurich Instruments LabOne/`

   to the `instr.lib` directory in your LabVIEW installation as located in Step 1. Note, you will need administrator rights to copy to this directory.

   Note, when copying from the main LabOne tarball (Linux only), the `Zurich Instruments LabOne/` directory is located in

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

3. Restart LabVIEW and verify your installation as described in Section 6.1.4.

6.1.4. Verifying your Installation

If the LabOne LabVIEW API palette can be accessed from within LabVIEW, the LabOne LabVIEW API is correctly installed. See Section 6.2.1 for help finding the palette.

6.2. Getting Started with the LabOne LabVIEW API

6.2.1. Locating the LabOne LabVIEW VI Palette

In order to locate the LabOne LabVIEW VIs start LabVIEW and create a new VI. In the VI's "Block Diagram" (CTRL-e) you can to access the LabOne LabVIEW API palette with a mouse right-click and browsing the tree under "Instrument I/O" → "Instr. Drivers", see Figure 6.1.

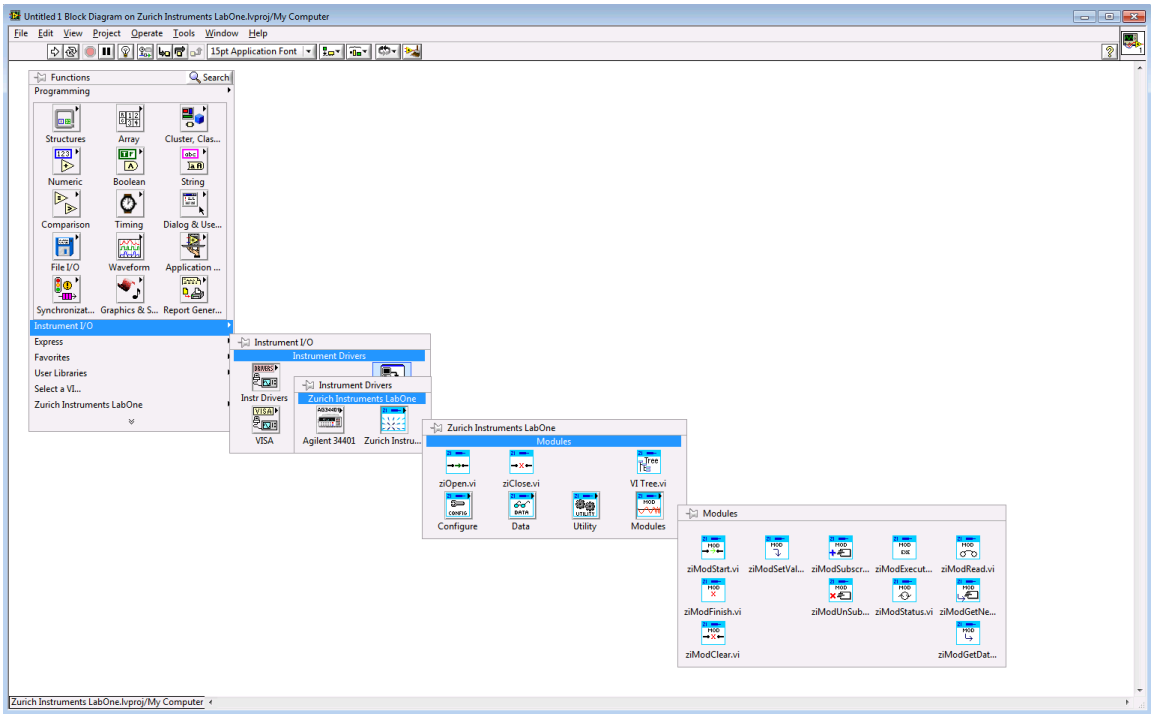


Figure 6.1. Locating the LabOne LabVIEW Palette

6.2.2. LabOne LabVIEW Programming Concepts

As described in Section 1.2 a LabVIEW program communicates to a Zurich Instrument device via a software program running on the PC called the data server. In general, the outline of the instruction flow for a LabVIEW virtual instrument is as following:

1. Initialization: Open a connection from the API to the data server program.
2. Configuration: Perform the instrument's settings. For example, using the virtual instrument `ziSetValueDouble.vi`.
3. Data: Read data from the instrument.
4. Utility: Perform data analysis on the read data, potentially repeating Step 2 and/or Step 3.
5. Close: Terminate the API's connection to the data server program.

The `VI Tree.vi` included the LabOne LabVIEW API demonstrates this flow and lists common VIs used for working with a Zurich Instruments device, see Figure 6.2. The `VI Tree.vi` can be found either via the LabOne VI palette, see Section 6.2.1, or by opening the file in the `Public` folder of your LabOne LabVIEW installation, typically located at:

```
C:\Program Files\National Instruments\LabVIEW 2012\instr.lib\Zurich Instruments
 LabOne\Public\VI Tree.vi
```

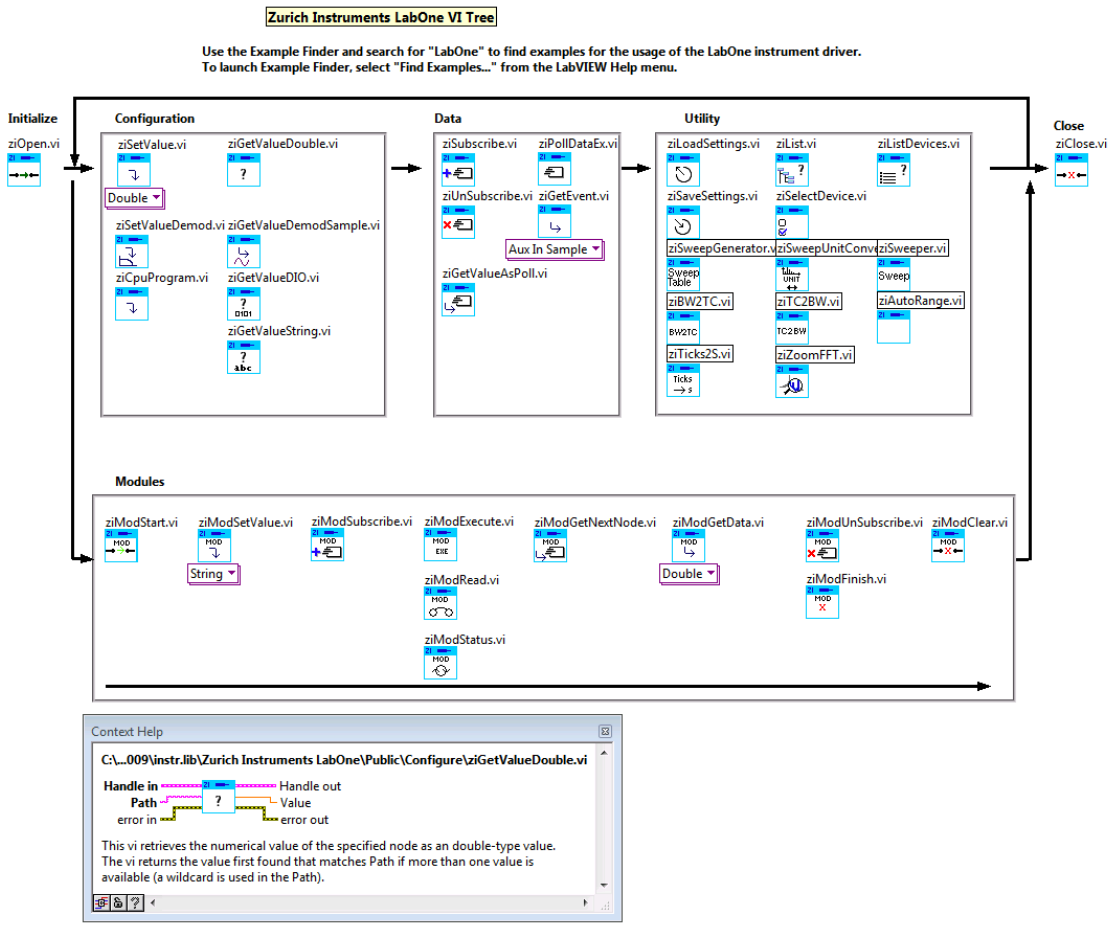


Figure 6.2. An overview of the LabOne LabVIEW VIs is given in `VI Tree.vi`. Press CTRL-h after selecting one of the VIs to obtain help.

6.2.3. Using ziCore Modules in the LabOne LabVIEW API

LabOne **ziCore** Modules Modules (e.g. Sweeper) enable high-level measurement tools to use with your Zurich instrument device in LabVIEW. The outline of the instruction flow for a LabVIEW Module is as following:

1. Initialization: Create a ziModHandle from a ziHandle **`ziModStart.vi`**.
2. Configuration: Perform the module's settings. For example, using the virtual instrument **`ziModSetValue.vi`**.
3. Subscribe: Define the recorded data node **`ziModSubscribe.vi`**.
4. Execute: Start the operation of the module **`ziModExecute.vi`**.
5. Data: Read data from the module. For example, using the **`ziModGetNextNode.vi`** and **`ziModGetData.vi`**.
6. Utility: Perform data analysis on the read data, potentially repeating Step 2, Step 3 and/or Step 4.
7. Clear: Terminate the API's connection to the module **`ziModClear.vi`**.

6.2.4. Finding help for the LabOne VIs from within LabVIEW

As is customary for LabVIEW, built-in help for LabOne's VIs can be obtained by selecting the VI with the mouse in a block diagram and pressing CTRL-h to view the VI's context help. See Figure 6.2 for an example.

6.2.5. Finding the LabOne LabVIEW API Examples

Many examples come bundled with the LabOne LabVIEW API which demonstrate the most important concepts of working with Zurich Instrument devices. The easiest way to browse the list of available examples is via the NI Example Finder: In LabVIEW select "Find Examples..." from the "Help" menu-bar and search for "LabOne", see Figure 6.3.

The examples are located in the directory `instr.lib/Zurich Instruments LabOne/Examples` found in LabVIEW installation directory. In order to modify an example for your needs, please copy it to your local workspace.

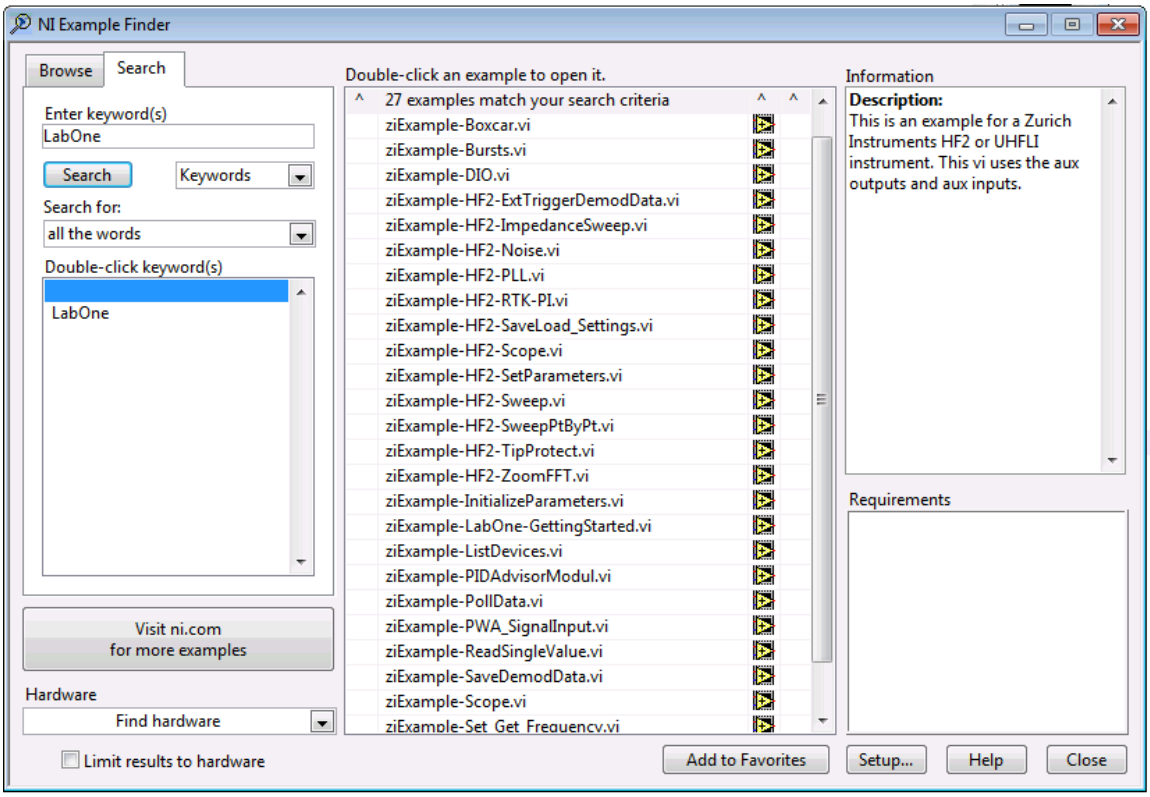


Figure 6.3. Search for "LabOne" in NI's Example Finder to find examples to run with your instrument.

6.2.6. Running the LabOne Example VIs

This section describes how to run a LabOne LabVIEW example on your instrument.

Note

Please ensure that the example you would like to run is supported by your instrument class and its options set. For example, examples for HF2 Instruments can be found in the Example Finder

(see Section 6.2.5) by searching for "HF2", examples for the UHFLI by searching for "UHFLI" and examples for the MFLI by searching for "MFLI".

Device Connection

After opening one of the LabOne LabVIEW examples, please ensure that the example is configured to run on the desired instrument type. **ziOpen.vi** establishes a connection to a Data Server. The address is of the format **{<host>}{:<port>}::{<Device ID>}**. Usually it is sufficient to provide the Device ID only highlighted in Figure 6.4. The Device ID corresponds to the serial number (S/N) found on the instrument rear panel. The host and port are then determined by network discovery. Should the discovery not work, prepend **<host>:<port>::** to the Device ID. Examples are "myhf2.company.com:8004::dev466" or "myhf2.company.com:8004". In the latter case the first found instrument on the data server listening on "myhf2.company.com:8004" will be selected.

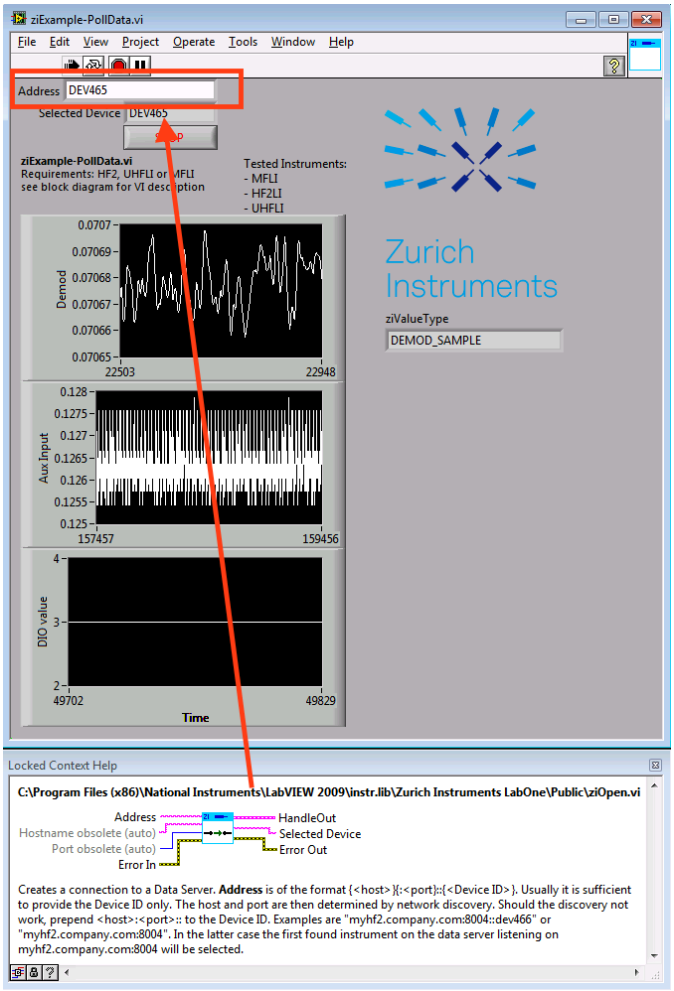


Figure 6.4. LabOne LabVIEW Example Poll Data: Device selection.

Running the VI and Block Diagram

The example can be ran as any LabVIEW program; by clicking the "Run" icon in the icon bar. Be sure to check the example's code and explanation by pressing CTRL-e to view the example's block diagram, see Figure 6.5.

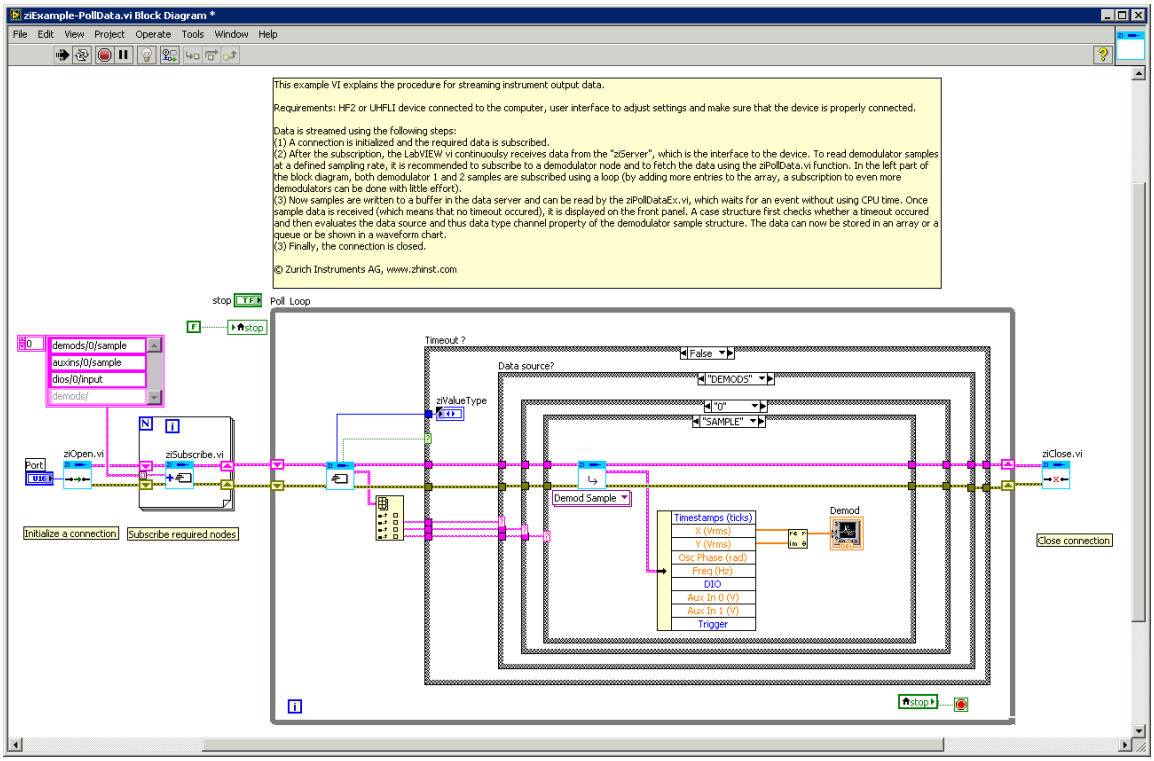


Figure 6.5. LabOne LabVIEW Example Poll Data: Block Diagram.

6.3. LabVIEW Programming Tips and Tricks

Use the User Interface's command log or Server's text interface while programming with LabVIEW

As with all other interfaces, LabVIEW uses the "path" and "nodes" concept to address settings on an instrument, see Section 1.2. In order to learn about or verify the nodes available it can be very helpful to view the command log in the User Interface (see the bar in the bottom of the screen) to see which node has been configured during a previous setting change. The text interface (HF2 Series) provides a convenient way to explore the node hierarchy.

Always close ziHandles and ziModHandles or LabVIEW runs out of memory

If you use the "Abort Execution" button of LabVIEW, your LabVIEW program will not close any existing connections to the ziServer. Any open connection inside of LabVIEW will persist and continue to consume about 12 MB of RAM so that with time you will run out of memory. Completely exit LabVIEW in order to release the memory again.

Use shift registers

The structure of efficient LabVIEW code is distinguished by signals being "piped through" by use of shift registers in loops and by the absence of object replication. Using shift registers in LabVIEW avoids copying of data and, more important, running the garbage collector frequently.

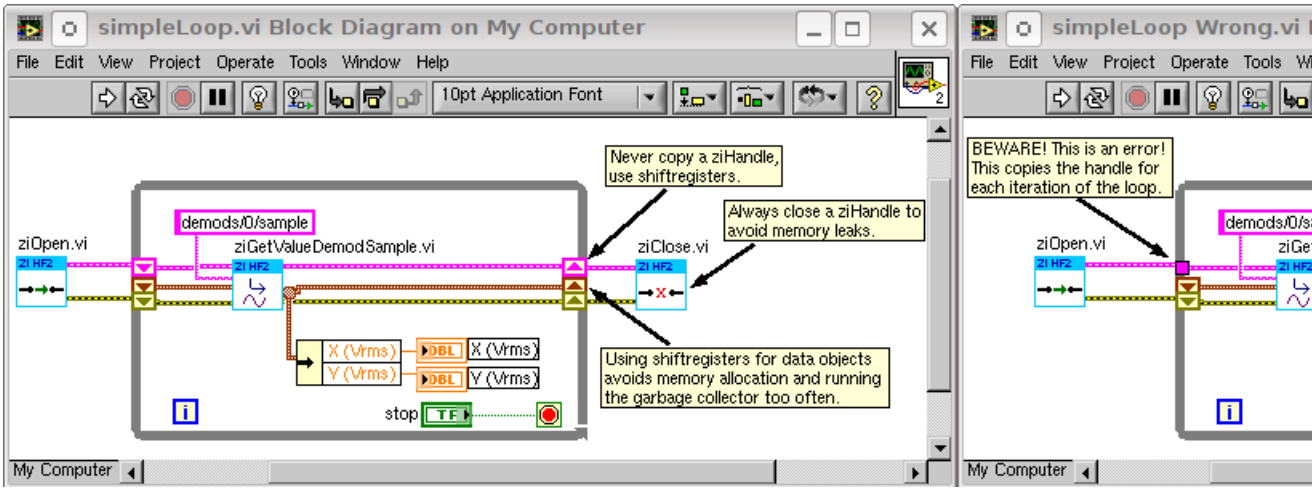


Figure 6.6. Examples of simple LabVIEW programs for the Zurich Instruments HF2 Series. Left: A well implemented loop, Right: An example while-loop implemented wrong.

Chapter 7. .NET Programming

This chapter helps you get started using Zurich Instruments LabOne's .NET API to control your instrument or integrate your instrument into an established .NET based control framework.

Please refer to:

- Section 7.1 for help installing the LabOne .NET API.
- Section 7.2 for help getting startedwith the LabOne .NET API.

LabOne API documentation

This chapter and the examples are not intended to be a .NET and Visual Studio or an introduction to any specific programming language.

Note

For a full reference of the .Net API visit the LabOne API documentation. The LabOne API documentation is available within your LabOne Software or can be accessed online at the Zurich Intruments website (www.zhinst.com)

7.1. Installing the LabOne .NET API

7.1.1. Requirements

To use LabOne's .NET API, **ziDotNET**, a Microsoft Visual Studio installation is required. The .NET API is a class library supporting x64 and win32 platforms. As the API is platform specific the project also needs to be platform specific.

The LabOne .NET API **ziDotNET** is included in a standard LabOne installation. No installation as such is required, but the corresponding dynamically linked library (DLL) files need to be copied to the folder of the Visual Studio solution, and a few configuration steps must be performed. The main LabOne installer is available from Zurich Instruments' download page (www.zhinst.com/downloads).

7.2. Getting Started with the LabOne .NET API

This section introduces the user to the LabOne .NET API. In order to use the LabOne API for .NET applications two DLL libraries should be copied to the application execution folder. The libraries are platform specific. Therefore, the project platform of the project should be restricted either to x64 or win32 CPU architecture. The following figures illustrate the initial steps to create a C# project using the LabOne API. The setup for other languages like Visual Basic or F# is equivalent.

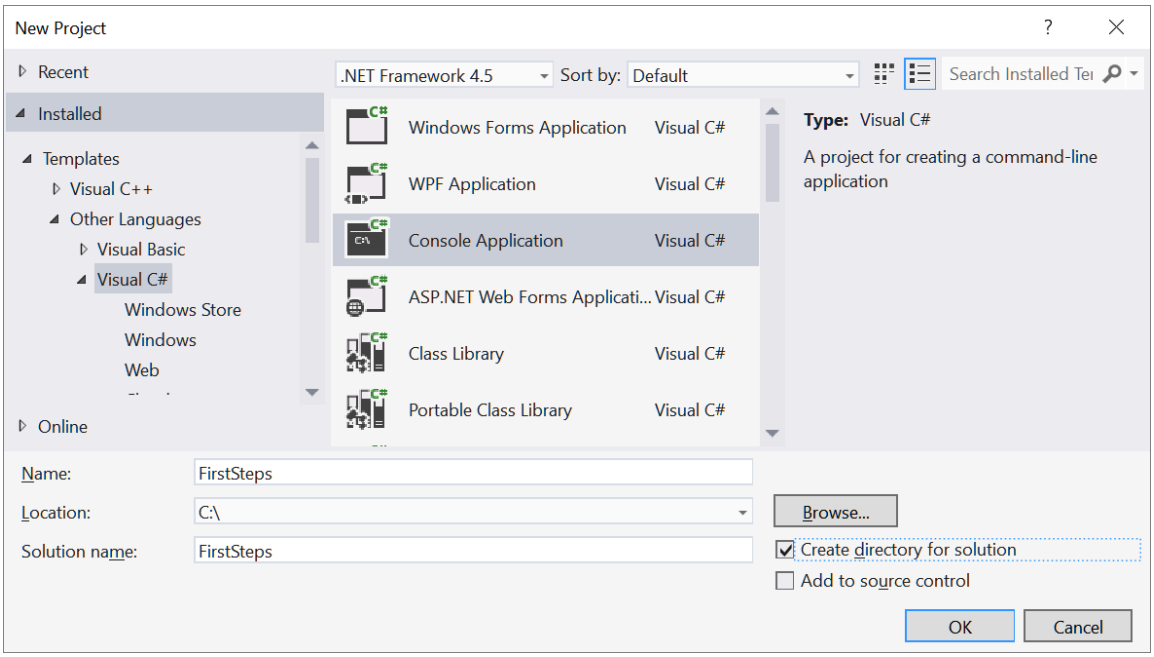


Figure 7.1. Creating a new C# project based on a solution.

Create a new project and choose Visual C# as a programming language and Console Application as target.

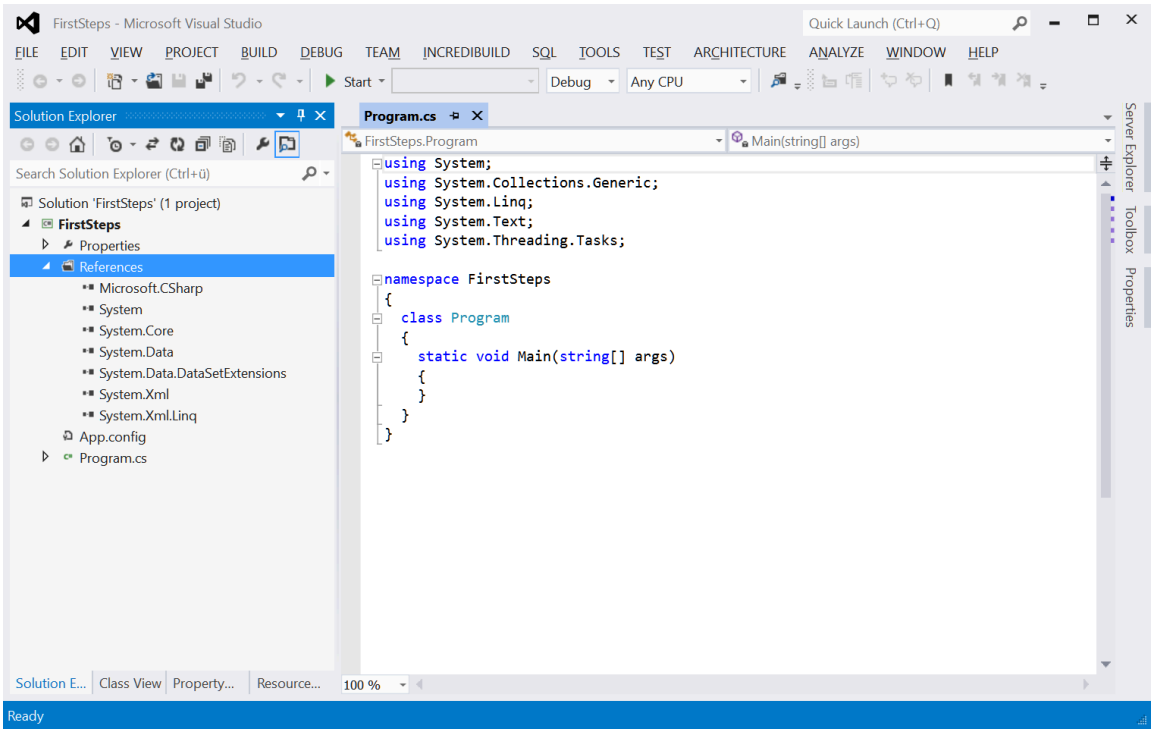


Figure 7.2. C# project with main program code opened in editor. Initially the project will support any CPU architecture. The ziDotNET API only supports x64 and win32 platforms.

The first step which needs to be done is to define the target platform as initially a Visual C# project is platform independent. To do this, click on the **Active solution platform** box, select **Configuration Manager…** to open the the **Configuration Manager**. In the following window click on the arrow under platform add a New target and choose x64 (Figure 7.3).

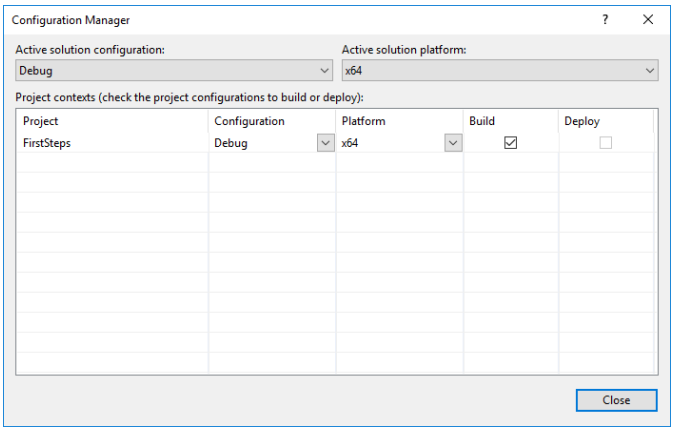


Figure 7.3. C# project with main program code opened in editor. Initially the project will support any CPU architecture. The ziDotNET API only supports x64 and win32 platforms.

The LabOne API for .NET consists of two DLLs for each platform that supply all functionality for connecting to the LabOne Data Servers on the specific platform (x64 and win32) and executing LabOne Modules. For simplicity we only discuss the x64 platform in this section, but the needed steps are analogous for the win32 platform. For x64 the two DLLs are ziDotNETCore-win64.dll and ziDotNET-win64.dll. The two DLL must accompany the executable using the functionality. The DLL files are installed under your LabOne installation path in the API/DotNet folder (usually **C:\Program Files\Zurich Instruments\LabOne\API\DotNET**). Copy the two DLLs for your platform into the solution folder.

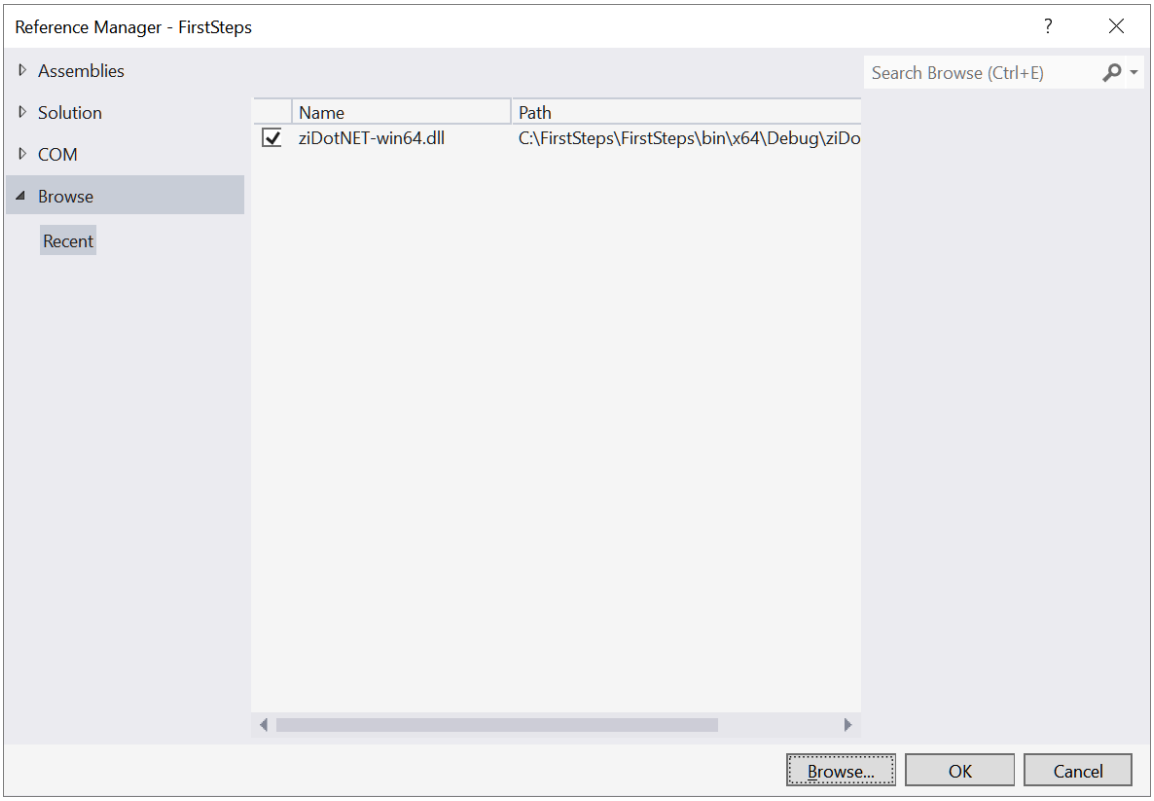


Figure 7.4. Reference to the API DLL ziDotNET for the specific platform.

To add the DLL to the project go to the solution explorer of your project (Figure 7.2) and right click on References and add the ziDotNET-win64.dll (Figure 7.4)

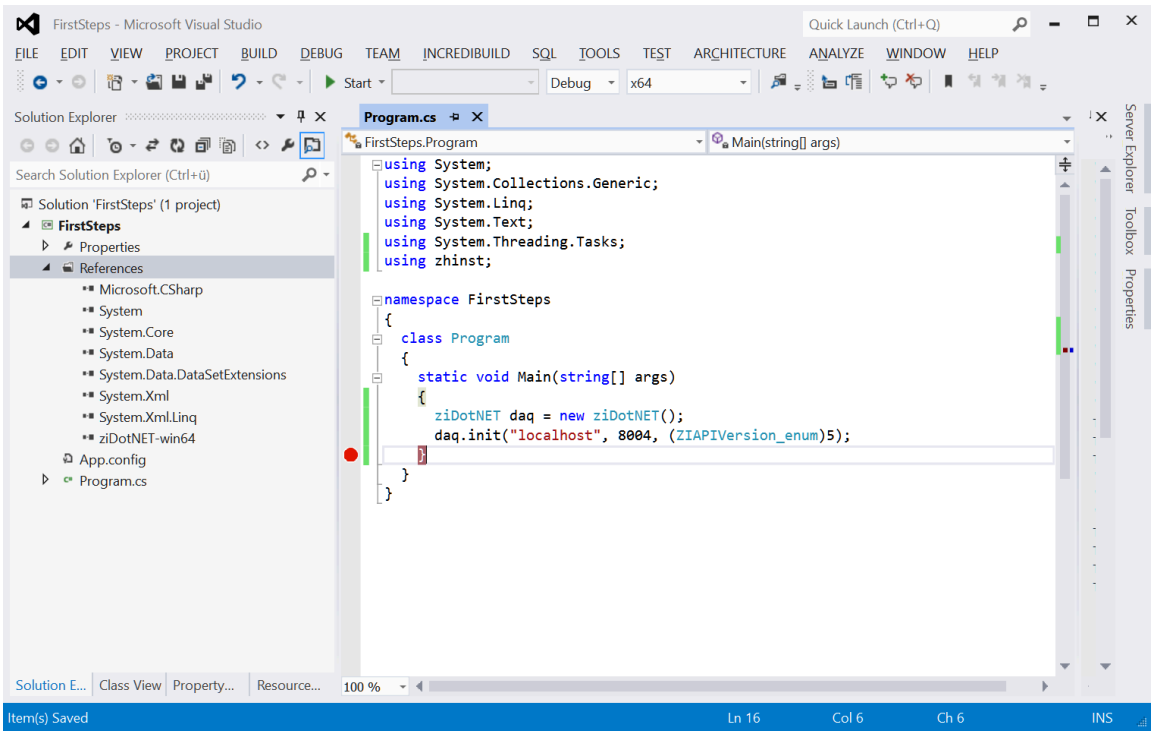


Figure 7.5. Added `using zhinst;` statement and code to open the connection to the server.

Figure 7.5 shows a first simple program which is done by adding `using zhinst;` to the include directive and the following code to the main body.

```
ziDotNET daq = new ziDotNET();
daq.init("localhost",8004,(ZIAPIVersion_enum)5);
```

If everything is configured correctly, the code compiles and when executed opens a session to a running LabOne data server and closes it before exiting the program.

7.3. LabOne .NET API Examples

The source code for the following program (Examples.cs) can be found in your LabOne installation path in the `API/DotNet` folder (usually `C:\Program Files\Zurich Instruments\LabOne \API\DotNET`).

The examples are documented in the LabOne API documentation.

Chapter 8. C Programming

The LabOne C API, also known as ziAPI, provides a simple and robust way to communicate with the Data Server. It enables you to get or set parameters and receive streaming data.

LabOne API documentation

For a full reference of the ziAPI visit the LabOne API documentation. The LabOne API documentation is available within your LabOne Software or can be accessed online at the Zurich Intruments website (www.zhinst.com)

8.1. Getting Started

After installing the LabOne software package and relevant drivers for your instrument you are ready start programming with ziAPI. All you need is a C compiler, linker and editor. The structure of a program using ziAPI can be split into three parts: initialization/connection, data manipulation and disconnection/cleanup. The basic object that is always used is the ziConnection data structure. First, ziConnection is has to be initialized by calling ziAPIInit. After initialization ziConnection is ready to connect to a ziServer by calling ziAPIConnect. Then ziConnection is ready to be used for getting and setting parameters and streaming data. When ziConnection is not needed anymore the established connection to the ziServer has to be hung up using ziAPIDisconnect before cleaning it up by calling ziAPIDestroy.

8.1.1. Examples

Along with the LabOne C API DLL, a LabOne installation includes examples to help getting started with the LabOne C API.

On Windows they are located in the folder:

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

and on Linux and macOS, after extracting the LabOne tarball, they are located in the folder:

`API/C/examples`.

Below you find a simple program, which sets the demodulator rate of all demods for all devices.

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

#include "ziAPI.h"

int main() {
  ZIResult_enum retVal;
  ZIConnection conn;
  char* errBuffer;
  const char* serverAddress = getenv("LABONE_SERVER");
  if (serverAddress == NULL) {
    serverAddress = "localhost";
  }
  printf("ENV LABONE_SERVER=%s\n", serverAddress);

  const char* device = getenv("LABONE_DEVICE_MF");
  if (device == NULL) {
    device = "dev3519";
  }
  printf("ENV LABONE_DEVICE_MF=%s\n", device);

  // Initialize ZIConnection.
  retVal = ziAPIInit(&conn);
  if (retVal != ZI_INFO_SUCCESS) {
    ziAPIGetError(retVal, &errBuffer, NULL);
    fprintf(stderr, "Can't init Connection: %s\n", errBuffer);
    return 1;
  }

  // Connect to the Data Server: Use port 8005 for the HF2 Data Server, use
  // 8004 for the UHF and MF Data Servers. HF2 only support ZI_API_VERSION_1,
  // see the LabOne Programming Manual for an explanation of API Levels.
  retVal = ziAPIConnectEx(conn, serverAddress, 8004, ZI_API_VERSION_6, NULL);
  if (retVal != ZI_INFO_SUCCESS) {
```

```
    ziAPIGetError(retVal, &errBuffer, NULL);
    fprintf(stderr, "Error, can't connect to the Data Server: `%s`.\n",
 errBuffer);
  } else {
    // Set all demodulator rates of device dev3519 to 150 Hz
    char path[1024];
    snprintf(path, sizeof(path), "/%s/demods/*/rate", device);
    retVal = ziAPISetValueD(conn, path, 150);
    if (retVal != ZI_INFO_SUCCESS) {
      ziAPIGetError(retVal, &errBuffer, NULL);
      fprintf(stderr, "Can't set parameter: %s\n", errBuffer);
    }

    // Disconnect from the Data Server. Since ZIAPIDisconnect always returns
    // ZI_INFO_SUCCESS no error handling is required.
    ziAPIDisconnect(conn);
  }

  // Destroy the ZIConnection. Since ZIAPIDestroy always returns
  // ZI_INFO_SUCCESS, no error handling is required.
  ziAPIDestroy(conn);

  return 0;
}
```

8.2. Error Handling and Logging in the LabOne C API

This section describes how to get more information when an error occurs.

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.

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 Modulated 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	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	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 (VCO)	Instrument that measures the network parameters of electrical networks, commonly expressed as s-parameters. For this purpose it measures the voltage of an input signal providing both amplitude (gain) and phase information. For this characteristic an older name was gain phase meter. See also Gain Phase Meter, Scalar Network Analyzer.

X

XML	Extensible Markup Language: Markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

Z

ZCtrl	Zurich Instruments Control bus
ZoomFFT	This technique performs FFT processing on demodulated samples, for instance after a lock-in amplifier. Since the resolution of an FFT depends on the number of point acquired and the spanned time (not the sample rate), it is possible to obtain very highly resolution spectral analysis.
ZSync	Zurich Instruments Synchronization bus

Index

Symbols

.NET, 174
  Comparison to other interfaces, 15
  Examples, 180
  Getting started, 176
  Installing the API, 175
  Requirements, 175

A

API
  Compatibility, 18
  Levels, 18
  Versions, 18
Asynchronous commands, 40
AWG Module, 46

C

C API
  Comparison to other interfaces, 15
C Programming Language
  Comparison to other interfaces, 15
Comparison of LabOne APIs, 15
Compatibility
  Data Server and API, 21
  Software, 21

D

Data Acquisition Module, 58
Data Server, 11
  Node, 23
Data Streaming, 31
Device Settings Module, 80
Device Synchronisation Module, 91

I

Impedance Module, 83

L

LabOne
  API overview, 15
  Comparison of APIs, 15
LabOne Programming
  Quick Start Guide, 8
LabVIEW, 165
  Comparison to other interfaces, 15
  Concepts, 168
  Examples, finding, 170
  Examples, running, 170
  Finding examples, 170
  Finding help, 170
  Getting started, 168
  Installation, 166
    Linux, 167
    Mac, 167
    Windows, 166
  LabOne VI Palette, 168
  Modules, 169
  Palette, LabOne, 168
  Requirements, 166
  Running examples, 170
  Tips and tricks, 173
  VI Palette, 168
Low-level commands, 41

M

MATLAB, 148
  Built-in help, 152
  Comparison to other interfaces, 15
  Contents of the API package, 152
  Examples, running, 152
  Getting started, 152
  Help, accessing, 152
  Installation, 149, 149
  Logging, 154
  Modules, 154
  Modules, configuring, 154
  Requirements, 149
  Running examples, 152
  Tips and tricks, 155
  Troubleshooting, 157
  Verifying correct configuration, 150
MF
  Data Server, 37
Multi-Device Synchronisation Module, 91
Multi-threading, 18

N

Node
  Leaf, 23
  Listing Nodes, 24
  Properties, 24
  Server node, 25
  Streaming nodes, 31
  Types, 24

P

PID Advisor Module, 94
PLL Module, 97
Precompensation Advisor Module, 108
Python, 159
  Built-in help, 162
  Comparison to other interfaces, 15
  Contents of the API package, 162
  Getting started, 162
  Help, accessing, 162
  Installation, 160
  Installing the API, 160
  Loading data in Python, 164

Logging, 163
Modules, 162
Modules, configuring, 162
Recommended python packages for the LabOne
Python API, 160
Requirements for using the Python API, 160
Tips and tricks, 164

Q

Quantum Analyzer Module, 119
Quick Start Guide
LabOne Programming, 8

S

Scope Module, 123
Streaming, 31, 31
Sweeper Module, 134
Bandwidth control, 134
Measurement data, 136
Measurement data, averaging, 136
Scanning mode, 134
Settling time, 135
Settling time, definition, 135
Sweep parameter, 134
Sweep range, 134
Synchronous commands, 40

U

UHF
Automatic calibration, 37
Calibration, 37

Z

ziAPI
Comparison to other interfaces, 15
ziDAQ
Installation, 149
Requirements, 149