LabOne API User Manual

Zurich
Instruments

LabOne API User Manual

Zurich Instruments AG

Revision 25.10.0.274

Table of Contents

1. LabOne API Documentation

1.1. Release 25.10.0.274

L1 LabOne

1.2. LabOne. All in One.

LabOne is the software package to interface between the test and measurement devices produced by Zurich Instruments and the applications carried out by the users. The package includes a browser-based graphical user interface (GUI) as well as application program interfaces (APIs) for Python, MATLAB®, LabVIEW™, .NET, and C/C++. It is a cross-platform software tool with support for Windows, macOS, and Linux operating systems. LabOne provides various tools and functionalities in the following three categories.

- Instrument control
- Data acquisition
- Signal processing

This user manual focuses on all APIs and covers all instrument types. For detailed GUI documentation, look at the respective device user manual.

1.3. Release Notes

For the changes included in the current and previous LabOne releases, see the Release Notes.

1.4. Getting started

This document is only a subset of the LabOne API User Manual. The full documentation is available online at https://docs.zhinst.com/labone-api/.

If you are limited to the sources provided by the LabOne software only, follow the installation steps in the next chapter and look at the dedicated API folder in your LabOne installation directory. Each API ships with a set of examples and reference documentation to get you started quickly.

1.5. More resources

- An extended set of application-specific examples is available for each LabOne API within the driver-specific package.
- For Python and MATLAB®, an open-source collection of examples can be found on GitHub.
- The open-source `zhinst-toolkit` Python API offers a broad set of examples on GitHub.

2. Installation

2.1. Installation of LabOne

In order to use any LabOne API a LabOne Data Server needs to run within the network. The LabOne Data Server is the Software component that enable the communication with one or multiple Zurich Instruments Devices.

The easiest way to install the complete LabOne stack is by using the LabOne installer. Please follow the LabOne Setup guide.

Note

If the LabOne Data Server is running on a remote server, e.g. a central server, LabOne does not need to be installed but only the used API sources.

2.2. Separate LabOne API sources

All LabOne API sources, except the Python API, are included in the LabOne installer. The default paths are the following:

- Windows `C:\Program Files\Zurich Instruments\LabOne\API`
- Linux `<extracted_tar_ball>/API`
- MacOS `/Applications/LabOne xx.xx.app/Contents/Resources/API`

In addition to the LabOne installer, all API sources can be downloaded directly from the Download Center.

Warning

The LabOne API and Data Server version must match. By default, all APIs will either return an error or show a warning when connecting to a data-server on a different version. For more information, see the page on LabOne API/data-server version compatibility.

It is also strongly recommended to use the latest available LabOne version.

2.3. LabOne API setup

Python

All Python package are available on pypi and can be installed with pip. All actively supported python packages (currently > python 3.7) are supported.

It is recommended to install the `zhinst` package which includes

- LabOne Core API package (`zhinst.core`)
- Utility package (`zhinst.utils`)
- LabOne Toolkit API package (`zhinst.toolkit`)

```
pip install zhinst
```

Note

Although the `zhinst` package contains all packages needed one can also install or update the individual packages directly without any downsides.

Tipp

If pip is not available to you you can download the `zhinst.core` wheel directly from the Download Center.

C/C++

The C API folder includes both the include file `include/ziAPI.h` and the library files `lib/*`.

MATLAB

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

- Windows with MATLAB R2009b or newer.
- Linux with MATLAB R2016b or newer.
- MacOS and MATLAB R2013b or newer.

No additional installation steps are required to use the MATLAB API; it's only necessary to add the folder containing LabOne's MATLAB Driver to MATLAB's search path. This is done by either of the following steps:

1. To only add the MATLAB API to the current matlab session execute the `ziAddPath` script, located in your LabOne installation or the extracted zip archive of the separate MATLAB API package (Download Center).
   ```
   >> ziAddPath;
   ```
2. Run the `pathtool` and click **Add with Subfolders**. Browse to the "MATLAB" directory, located in your LabOne installation or the extracted zip archive of the separate MATLAB API package (Download Center), and click "OK".
3. Edit your `startup.m` to run the `ziAddPath` script, located in your LabOne installation or the extracted zip archive of the separate MATLAB API package (Download Center).
   ```
   >> run('<path to the MATLAB API sources>/ziAddPath');
   ```
   For more help on MATLAB's startup.m file, type the following in MATLAB's Command Window:
   ```
   >> docsearch('startup.m')
   ```

To verify that Matlab is aware of the LabOne API one can open the help of the `ziDAQ` object, which is the entry point into the LabOne MATLAB API.

```
>> help ziDAQ
```

LabVIEW

One of the following platforms and LabVIEW versions is required to use the LabOne LabVIEW API:

- Windows with LabVIEW 2009 or newer.
- Linux with LabVIEW 2010 or newer.
- MacOS and LabVIEW 2010 or newer.

In order to make the LabOne LabVIEW API available for use within LabVIEW, the driver directory needs to be copied to a specific directory of your LabVIEW 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:
   - Windows: `C:\Program Files\National Instruments\LabVIEW 201x\instr.lib\`
   - Linux: `/usr/local/natinst/LabVIEW-201x/instr.lib/`
   - MacOS: `/Applications/National Instruments/LabVIEW 201x/instr.lib/`
2. Copy the directory `Zurich Instruments LabOne` from your in your LabOne installation (API/LabVIEW) or the extracted zip archive of the separate LabVIEW API package (Download Center) 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.
3. Restart LabVIEW. The `Zurich Instruments LabOne` pallet should now be available under `Instrument I/O` → `Instr. Drivers`

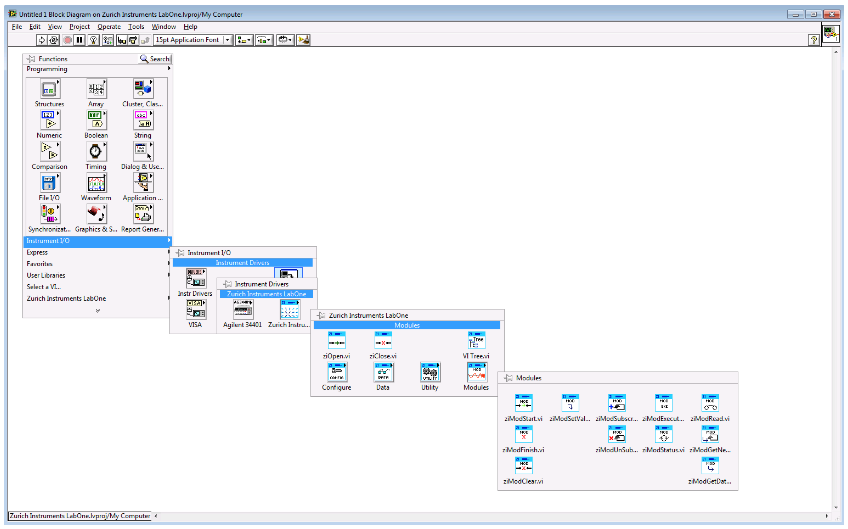
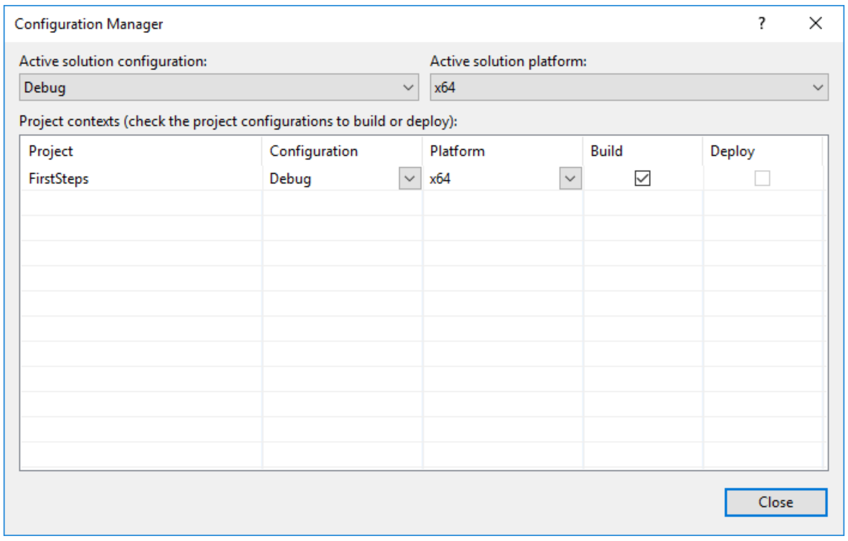


Figure 1.1: Image title

.NET

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. Therefore, the project platform of the project should be restricted either to x64 or win32 CPU architecture. 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 the right platform.



Configuration Manager

Active solution configuration: Debug

Active solution platform: x64

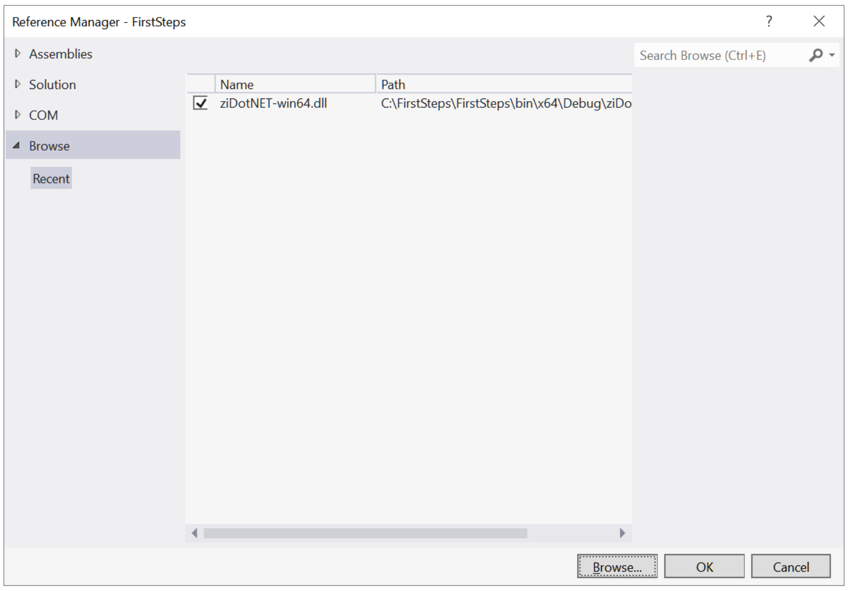
Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
FirstSteps	Debug	x64	✓	☐

Figure 1.2: front panel

For simplicity we only show the images for x64 platform, 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.

To add the DLL to the project go to the solution explorer of your project and right click on References and add the ziDotNET-win64.dll



	Name	Path
✓	ziDotNET-win64.dll	C:\FirstSteps\FirstSteps\bin\x64\Debug\ziDo

Figure 1.3: front panel

Troubleshooting

In case of problems contact us via support@zhinst.com.

3. LabOne Modules

In addition to the usual API commands available for instrument configuration and data retrieval, e.g., `setInt`, `poll`), the LabOne API 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 LabOne API directly as a separate software component.

Info

The different tabs in the LabOne UI are often based directly on a LabOne Module. The behaviour of the LabOne UI can therefore often be directly mapped to the LabOne API.

Important

It is possible to create more than one instance of a LabOne Module. Each instance creates a separate session to the connected Data Server and therefore does not disturb other modules or the main session.

Since each session consumes on both sides resources it is recommended to remove unused modules and try to keep the amount of modules to a minimum.

The following modules are available:

Module	Description
Sweeper	Obtaining data whilst performing a sweep of one of the instrument's setting, e.g., measuring a frequency response.
Data Acquisition	Recording instrument data <> based upon user-defined triggers.
Device Settings	Saving and loading instrument settings to and from (XML) files.
PID Advisor	Modeling and simulating the PID incorporated in the instrument.
Scope Module	Obtaining scope data from the instrument.
Impedance	Performing impedance measurements.
Multi-Device Synchronisation	Synchronizing the timestamps of multiple instruments.
AWG	Working with the AWG.
Precompensation Advisor	Working with the AWG.

3.1. Usage

LabOne Modules are, similar to the devices, controlled mainly through nodes. Each module has a defined list of available nodes. The API of the modules offer the same set of functions to manipulate and read the nodes.

Note

An important difference to Data Server communication is that the LabOne Modules live directly in the Client. Therefore changing node values do not require any network communication and take effect immediately. This is also the reason why there is only a single type of commands for settings and getting node values.

In addition to the nodes all LabOne Modules have additional commands :

- **execute** command to start the module execution.
- **finish** command to stop the execution.
- **finished** command to check if the execution has finished.
- **progress** command to check the progress of the execution with a number between 0 and 1.
- **read** command to read the module output data.
- **save** command to save measured data to a file.
- **trigger** command to execute a manual trigger.

Note

Although all LabOne Modules have the same additional commands, not all of them make use of all of the commands. Some commands, e.g. the `trigger` command, might not be used, but still are available, in all modules.

3.2. 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.

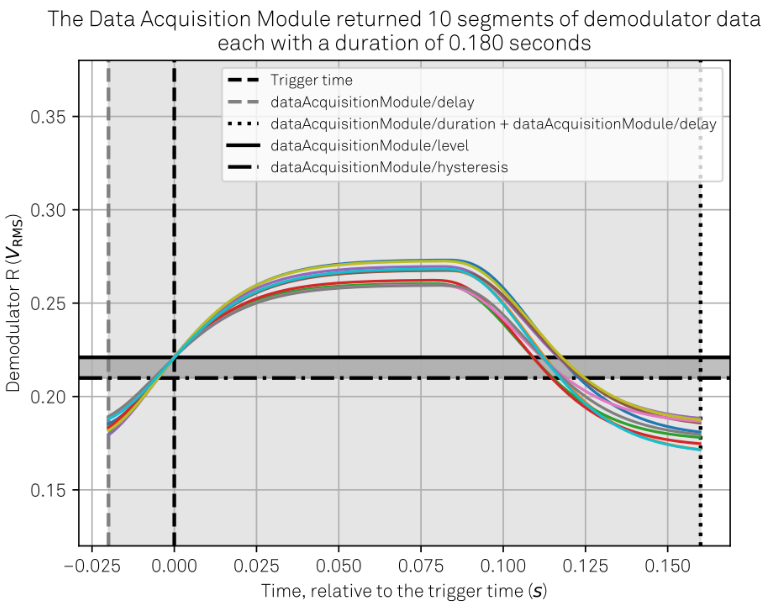


Figure 2.1: example_data_acquisition_edge

3.2.1. 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 Data Acquisition Module Node Tree.

The following acquisition modes exists:

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
Tracking (Edge or Pulse)	Level tracking trigger to compensate for signal drift.	4

Mode / Trigger Type	Description	Value of type
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 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.

The following image explains the Data Acquisition Module's parameters for an Edge Trigger.

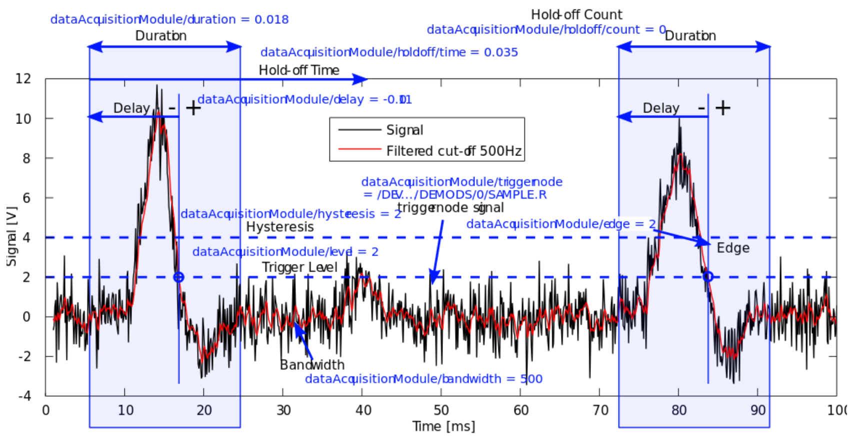


Figure 2.2: daq_analog_trigger

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 for more information.

The following image explains the Data Acquisition Module's parameters for a positive Pulse Trigger.

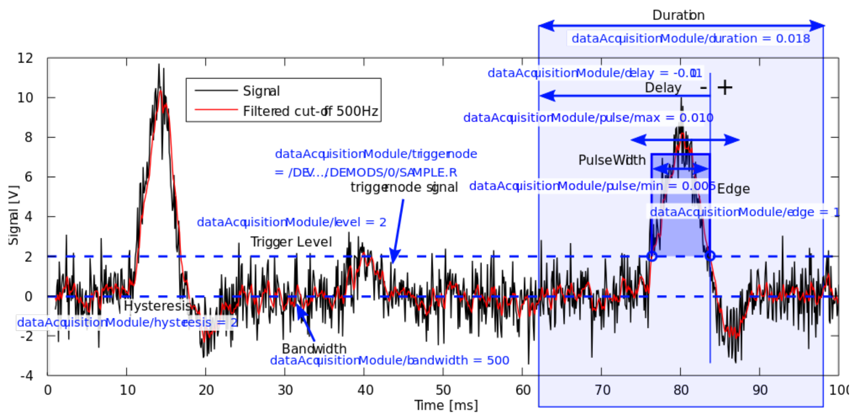


Figure 2.3: daq_pulse_trigger

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`

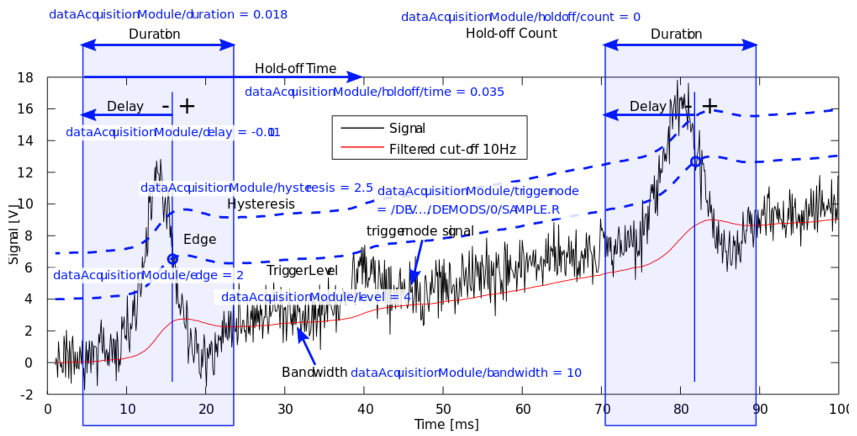


Figure 2.4: daq_tracking_trigger

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`.

3.2.2. 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. 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.

The following python code demonstrates how to use the `/findlevel` parameter.

```python
# 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')
```

3.2.3. Signal Subscription

The Data Acquisition Module uses dot notation for subscribing to the signals. Whereas with the Software Trigger (Recorder Module) you subscribe to an entire streaming node, e.g. `/DEV..../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):

Signal	Description
`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]
```

source_signal

Note

Nodes not listed here probably only contain one signal and do not have a source_signal field.

Signal Name	Description (Path of the node containing the signal(s))	Comment
`<demod>.x`	Demodulator output in-phase component	
`<demod>.y`	Demodulator output quadrature component	
`<demod>.r`	Demodulator output amplitude	
`<demod>.theta`	Demodulator output phase	
`<demod>.frequency`	Oscillator frequency	
`<demod>.auxin0`	Auxiliary input channel 1	
`<demod>.auxin1`	Auxiliary input channel 2	
`<demod>.xiy`	Combined demodulator output in-phase and quadrature components	complex output (can only be used as FFT input))

Signal Name	Description (Path of the node containing the signal(s))	Comment
`<demod>.df`	Demodulator output phase derivative (can only be used for FFT(dθ/dt)/(2π)	
`<impedance>.realz`	In-phase component of impedance sample	
`<impedance>.imagz`	Quadrature component of impedance sample	
`<impedance>.absz`	Amplitude of impedance sample	
`<impedance>.phasez`	Phase of impedance sample	
`<impedance>.frequency`	Oscillator frequency	
`<impedance>.param0`	Measurement parameter that depends on circuit configuration	
`<impedance>.param1`	Measurement parameter that depends on circuit configuration	
`<impedance>.drive`	Amplitude of the AC signal applied to the device under test	
`<impedance>.bias`	DC Voltage applied to the device under test	
`<impedance>.z`	Combined impedance in-phase and quadrature components	complex (can only be used as FFT input)

complex_selector (mandatory with `.fft`)

Signal Name	Description
`real`	Real component of FFT
`imag`	Imaginary component of FFT
`abs`	Absolute component of FFT
`phase`	Phase component of FFT

filter (optional)

Helper signal representing demodulator low-pass filter transfer function. It can only be applied to 'abs' FFT output of complex demodulator source signal, i.e. 'xiy.fft.abs.filter'. No additional operations are permitted. Can be used to compensate the FFT result for the demodulator low-pass filter.

pwr (optional)

Power calculation

math_operation (optional)

Signal Name	Description
avg	Average of grid repetitions (parameter grid/repetitions)
std	Standard deviation

3.2.4. Node Documentation

This section describes all the nodes in the Data Acquisition Module node tree organized by branch.

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 grid frames 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. dev1000 (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.

Value	Description
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).

Value	Description
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/powercompensation

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

Apply power correction to the spectrum to compensate for the shift that the window function causes.

/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.

Value	Description
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.

Value	Description
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.

Value	Description
1	"nearest": Use the closest data point (nearest neighbour interpolation).
2	"linear": Use linear interpolation.
4	"exact": 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.

Value	Description
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.

Value	Description
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).
9	"burst_trigger": Burst trigger (device triggered acquisition).

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 of the instruments from any of the LabOne APIs.

Important

The AWG module is the underlying logic of the LabOne User Interface AWG tab. However most of the LabOne APIs expose the AWG offline compiler directly. Often this is the preferred way and offer much more flexibility.

This page only explains the specifics for working with the LabOne AWG module. A in-depth documentation of the LabOne AWG Sequencer Programming Language and its usage can be found in the Sequencer Programming User Manual.

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 Core of the used 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.

3.3.1. Compilation

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

- Source file: In this case, the sequencer program file must reside in the `/awg/src` sub-directory of the LabOne user directory (The exact location of this directory can be read from the module node `/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.
- 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.

3.3.2. 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. Node Documentation

This section describes all the nodes in the AWG Module node tree organized by branch.

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/extrajson

Properties:	Read
Type:	String
Unit:	None

Extra compiler output (e.g., waveform memory usage) as JSON.

/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

Value	Description
-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. 'dev1000'.

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.

Value	Description
-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").

Value	Description
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. 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.

Important

In general, users are recommended to use the utility functions provided by the APIs instead of using the Device Settings module directly. 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).

3.4.1. Node Documentation

This section describes all the nodes in the Device Settings Module node tree organized by branch.

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. 'dev1000,dev2000'.

errortext

/errortext

Properties:	Read
Type:	String
Unit:	None

The error text used in error messages.

fileformat

/fileformat

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

The format of the settings file.

Value	Description
0	"xml": XML
1	"json": JSON

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.

3.5. 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.

3.5.1. Node Documentation

This section describes all the nodes in the Impedance Module node tree organized by branch.

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/n/c

Properties:	Read, Write
Type:	Double
Unit:	F

Parallel capacitance of the first compensation load (SHORT).

/loads/n/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.

Value	Description
0	"none": None
1	"short": S (Short)
2	"open": O (Open)
3	"short_open": SO (Short-Open)
4	"load": L (Load)
5	"short_load": SL (Short-Load)
6	"open_load": OL (Open-Load)
7	"short_open_load": SOL (Short-Open-Load)
8	"load_load_load": LLL (Load-Load-Load)

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.

Value	Description
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.

Value	Description
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.

3.6. 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 synchronisation, and also details the cabling required to achieve this.

3.6.1. Node Documentation

This section describes all the nodes in the Multi Device Synchronization Module node tree organized by branch.

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.

Value	Description
-1	error
0	idle
1	synchronization in progress
2	successful synchronization

3.7. 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 the following figure.

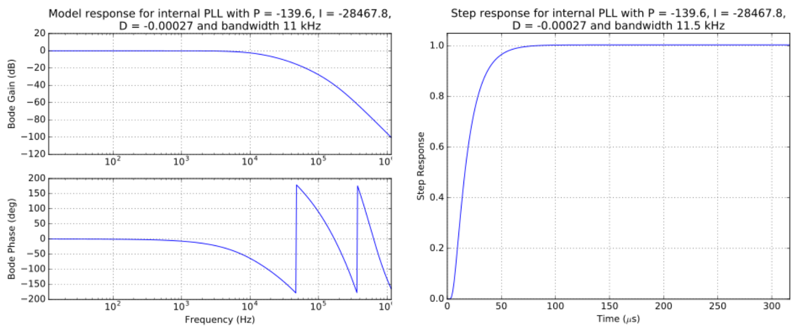


Figure 2.5: example_pid_advisor_pll_bode_step_combined

3.7.1. 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 node documentation 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 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.

3.7.2. 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.

3.7.3. 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 previous section for an explanation). For all other devices `/type` can only be set to "pid". The following tables describe which device nodes are configured.

The following device nodes are 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

The following additional device nodes configured when `/type` is "pid" (default behavior) and `/dut/source` = 4 (internal PLL).

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

The following device nodes configured when `/type` is "pll" (HF2 instruments only).

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.

3.7.4. 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.

Non HF2 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.

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.

3.7.5. Node Documentation

This section describes all the nodes in the PID Advisor Module node tree organized by branch.

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.

Value	Description
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

3.8. 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.

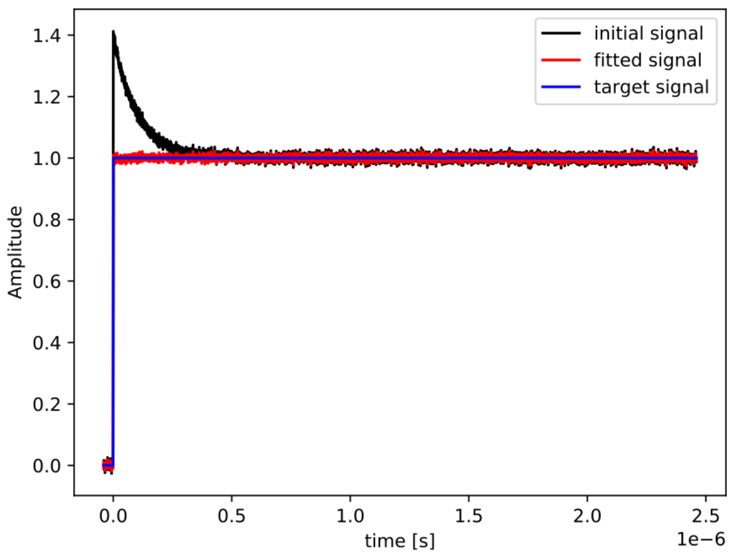


Figure 2.6: example_precompensation

3.8.1. 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.

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.

3.8.2. Node Documentation

This section describes all the nodes in the Precompensation Advisor Module node tree organized by branch.

bounces

/bounces/n/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the bounce compensation filter.

/bounces/n/delay

Properties:	Read, Write
Type:	Double
Unit:	None

Delay between original signal and the bounce echo.

/bounces/n/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/n/amplitude

Properties:	Read, Write
Type:	Double
Unit:	None

Amplitude of the exponential filter.

/exponentials/n/enable

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

Enable the exponential filter.

/exponentials/n/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/n/enable

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

Enable the high-pass compensation filter.

/highpass/n/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.

Value	Description
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.

3.9. 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.

- Shift or translation
- Rotation
- 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 the following equation:

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

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.

3.9.1. Node Documentation

This section describes all the nodes in the Quantum Analyzer Module node tree organized by branch.

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.

Value	Description
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.

3.10. Scope Module

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

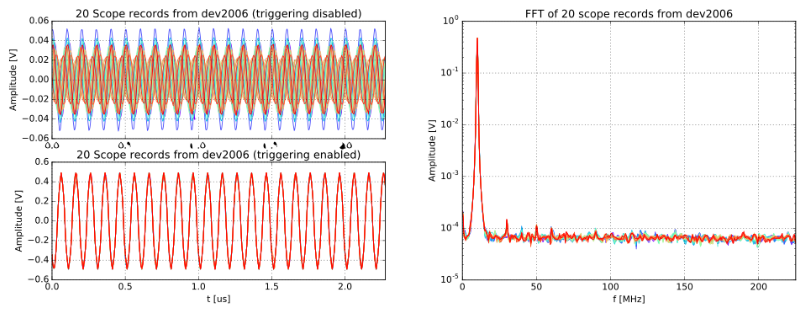


Figure 2.7: example_scope_module_freq_time_combined

3.10.1. Introduction to Scope Data Transfer

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

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

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

Segmented Mode

Additionally, MF and UHF instruments which have the DIG option enabled can optionally record data in "segmented" mode which allows back-to-back measurements with very small hold-off times between measurements. Segmented mode enables the user to make a fixed number of measurements (the segments), which are stored completely in the memory of the instrument, before they are transferred to the Data Server (this overcomes the data transfer rate limitation of the device's connected physical interface, e.g., USB or 1GbE for a fixed number of measurements). The advantage to this mode is precisely that the hold-off time, i.e. the delay between two measurements, can be very low. Data recorded in segmented mode is still available from the instrument node `/DEV..../SCOPES/n/WAVE` as for non-segmented data, but requires an additional reshaping described in Segmented Recording.

3.10.2. 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.

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.

3.10.3. 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 using the `averager/` parameters.
6. Can be configured to return a specific number of scope records using the `historylength` parameter.

3.10.4. Working with Scope Module

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

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

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

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

4. Configure the Scope Module's parameters as required, in particular: (a) Set `mode` in order to specify whether to return time or frequency domain data. See Scope Module Modes for more information on the Scope Module's modes. (b) 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. (c) Set the `averager` parameters to select the averaging method and enable averaging of scope records, see Averaging.
5. Enable the scope (if not previously enabled) and call Scope Module `execute()` to start acquiring data.
6. Wait for the Scope Module to acquire the specified number of records. Note, if certain scope parameters are modified during recording, the history of the Scope Module will be cleared; see Scope Parameters that reset the Scope Module for the list of parameters that trigger a Scope Module reset.
7. 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`.
8. Check the flags of each record indicating whether any problems occurred during acquisition.
9. 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 Segmented Recording) Note, the scope data structure only returns data for enabled scope channels.

Data Acquisition and Transfer Speed

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

3.10.5. 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`. These can be constructed as arrays of `N` points, where `N` is the scope length configured through the node `/DEV.../SCOPES/n/LENGTH` spanning the intervals:

Time mode

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

FFT mode

`[0, (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`.

3.10.6. Averaging

The Scope Module of LabOne offers an averaging feature to improve the signal-to-noise ratio (SNR) of the acquired signals by reducing the noise through averaging over multiple scope segments. The averaging engine starts by setting the module node `averager/enable` to 1. The module offers two averaging methods, namely Uniform and Exponential, which can be selected through `averager/method`. The uniform averaging implements the following recursive expression:

$$\begin{aligned} y[n] &= \frac{1}{n+1} \times x[n] + \frac{n}{n+1} \times y[n-1], \\ y[0] &= x[0], \end{aligned} \tag{1}$$

where x[n] is the n -th acquired scope segment, and y[n] is the averaged signal after n scope segments. The exponential averaging is based on the following recursive equation:

$$
\begin{aligned}
&y[n] = \frac{2}{w+1} \times x[n] + \frac{w-1}{w+1} \times y[n-1],\\
&y[0] = x[0],
\end{aligned}
\tag{2}
$$

in which w is the weight of exponential averaging and can be adjusted through the node `averager/weight`. The weight corresponds to the number of segments to achieve 63% settling; doubling the value of weight achieves 86% settling.

If the scope is in single acquisition mode, no averaging is performed. To obtain the number of segments that have been averaged, one can query the read-only node `averager/count`. The average calculation can be restarted by setting `averager/restart` to 1. This node will be automatically set to 0 when averaging starts.

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.

3.10.7. Segmented Recording

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

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

3.10.8. 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`

3.10.9. Scope Module Use on HF2 Instruments

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

3.10.10. Node Documentation

This section describes all the nodes in the Scope Module node tree organized by branch.

averager

/averager/count

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

Indicates the number of scope shots that have been averaged.

/averager/enable

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

Enable averaging. = 0: Averaging disabled. 1: Averaging enabled, updating last history entry, using the method set by averager/method.

/averager/method

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

Specify the averaging method.

Value	Description
0	"exponential": Exponential moving average
1	"uniform": Uniform averaging

/averager/resamplingmode

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

Specify 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.

Value	Description
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 weight of the exponential averaging method. It is not used by the uniform averaging method. Weight values 0 and 1 correspond to no averaging.

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/powercompensation

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

Apply power correction to the spectrum to compensate for the shift that the window function causes.

/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

Value	Description
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.

Value	Description
0	"passthrough": Passthrough: scope segments assembled and returned unprocessed, non-interleaved.
1	"exp_moving_average": Cumulative averaging: entire scope recording assembled, scaling applied, averaging if enabled (see averager/enable) using method set by averager/method, 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.

Value	Description
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.

3.11. Sweeper Module

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

3.11.1. Configuring the Sweeper

In this section we briefly describe how to configure the Sweeper Module. See the node documentation 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., `/DEV2345/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.

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 <> for recommended values of `settling/inaccuracy`. The relationship between `settling/inaccuracy` and `settling/tc` is the following:

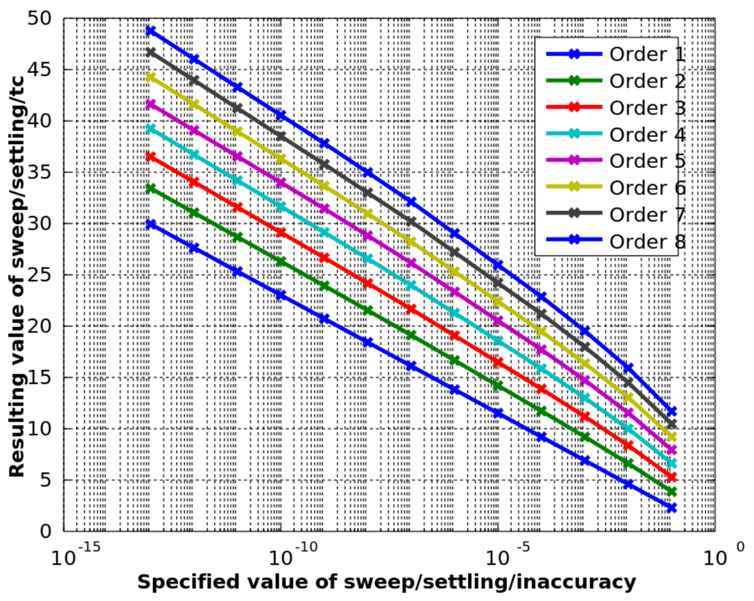


Figure 2.8: settling/inaccuracy - settling/tc relationship

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).

$$t_s = \max\{\text{tc} \times (\text{settling/tc}), (\text{settling/time})\} \tag{1}$$

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).

Specifying which Data to Measure

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

Specifying how the Measurement Data is Averaged

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

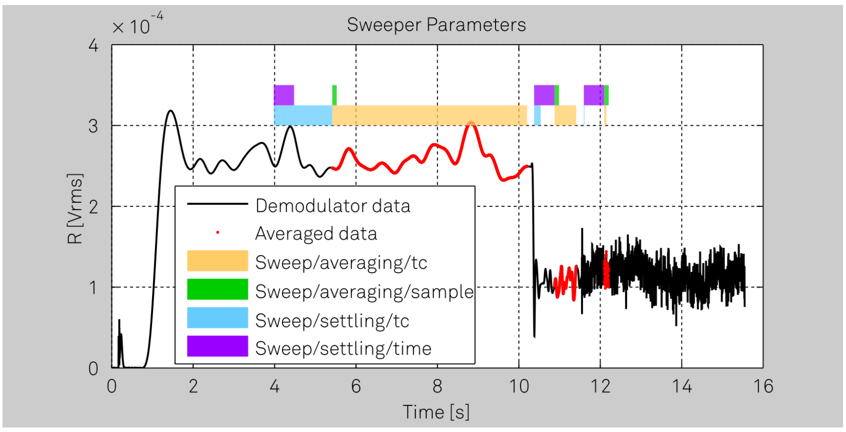
The number of samples N used to average one sweeper measurement point is determined by the parameters **`averaging/time`**, **`averaging/tc`**, and **`averaging/sample`** as well as the **`rate`** of data transfer from the instrument to the data server (see Equation 2).

$$N = \max\{\text{tc} \times (\text{averaging/tc}) \times \text{rate}, (\text{averaging/time}) \times \text{rate}, (\text{averaging/sample})\} \tag{2}$$

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 respective section above. 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



The image 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.

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.

3.11.2. Node Documentation

This section describes all the nodes in the Sweeper Module node tree organized by branch.

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.

Value	Description
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., dev1000 (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.

Value	Description
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 set on the device in Fixed and Auto bandwidth modes (sweep/ bandwidthcontrol=1 and 2). It ranges from 1 (6 dB/octave) to 4 (24 dB/octave) or 8 (48 dB/octave) depending on the device type.

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.

Value	Description
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.

Value	Description
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 derived from sweep/settling/inaccuracy. The maximum between this value and sweep/settling/time is taken as effective settling time. Note, although it is recommended to use sweep/settling/inaccuracy, it is still possible to set sweep/settling/tc directly (the parameter applied will be simply the last one set).

/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.

startdelay

/startdelay

Properties:	Read, Write
Type:	Double
Unit:	Seconds

Sets the wait time in seconds between setting the first sweep parameter value and the start of the measurement.

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).

Value	Description
0	"linear": Linear
1	"log": Logarithmic distribution of sweep parameter values