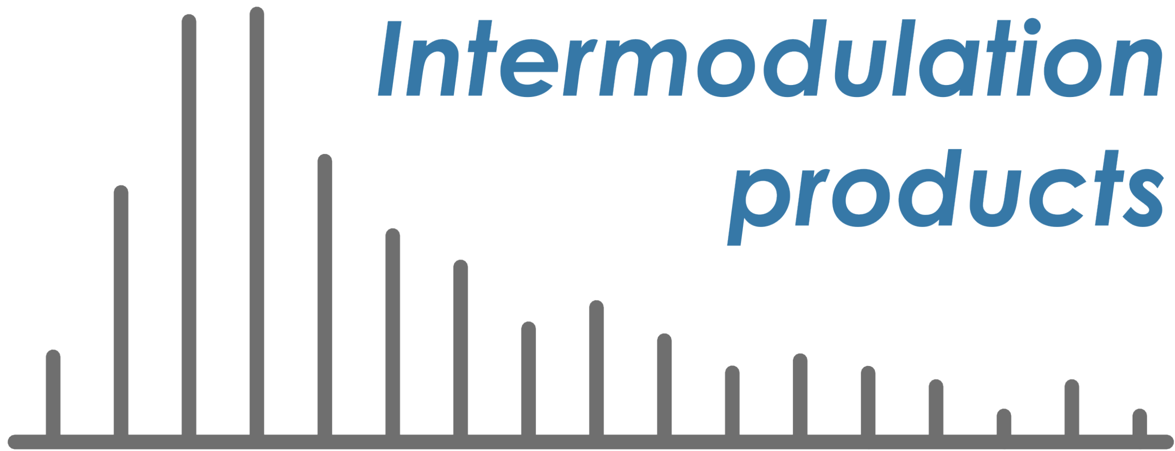


IMP Multifrequency Lockin User Manual

Release 4.7

Intermodulation Products AB

Dec 23, 2025

CONTENTS

1 Multifrequency Lockin Amplifier 3
1.1 connections MLA-3 . . . 3
1.1.1 front-side . . . 4
1.1.2 back-side . . . 4
1.1.3 grounding . . . 5

2 Quick Start Guide 7

3 Software installation 9
3.1 “exe” version . . . 9
3.2 “zip” version . . . 9

4 Multifrequency Lockin Basics 11
4.1 lockin measurement . . . 11
4.2 demodulating multiple frequencies . . . 12
4.3 Fourier leakage . . . 12
4.4 tuning and nonlinearity . . . 14
4.5 setup and tuning functions . . . 14
4.6 tuning frequency combs . . . 15
4.7 advanced lockin topics . . . 16
4.7.1 finite accuracy and perfect tuning . . . 16
4.7.2 frequency sweep . . . 17
4.7.3 triggered lockin measurement . . . 17
4.7.4 real-time feedback . . . 18
4.7.5 demodulation details . . . 18

5 Graphical User Interface (GUI) 19
5.1 common features . . . 19
5.2 analog panel . . . 20
5.3 aux output panel . . . 21
5.4 lockin setup panel . . . 21
5.5 oscilloscope panel . . . 22
5.6 lockin history panel . . . 22
5.7 script panel . . . 23
5.8 script plot panel . . . 23
5.9 frequency sweep panel . . . 23
5.10 frequency counter panel . . . 24
5.11 message log panel . . . 24
5.12 Python shell panel . . . 24

6 Communicating with the MLA™ 25
6.1 data streams . . . 25
6.2 streaming lockin data . . . 25
6.2.1 lockin data formats . . . 26
6.2.1.1 IQ-real . . . 26

6.2.1.2 IQ-complex . . . 26
6.2.1.3 amp and phase . . . 26
6.2.1.4 data conversion . . . 27
6.3 streaming time data . . . 27

7 MLA programming 29
7.1 NumPy and matplotlib . . . 29
7.2 MLA API . . . 30
7.2.1 mla . . . 30
7.2.2 mla.lockin . . . 32
7.2.3 mla.osc . . . 67
7.2.4 mla.analog . . . 73
7.2.5 mla.arb . . . 76
7.2.6 mla.feeback . . . 78
7.2.7 mla.hardware . . . 81
7.3 MLA GUI . . . 92
7.3.1 mla_gui . . . 93
7.3.2 Script panel . . . 94
7.3.3 Scripting utilities . . . 95
7.3.4 Shell panel . . . 95
7.3.5 Lockin setup panel . . . 96
7.3.6 Oscilloscope panel . . . 96
7.3.7 Lockin plot panel . . . 97
7.3.8 Line colors . . . 98
7.4 example scripts . . . 99
7.4.1 setup the GUI . . . 99
7.4.2 plotting in the GUI . . . 99
7.4.3 configuring the lockin . . . 100
7.4.4 lockin measurement loop . . . 100
7.4.5 frequency sweep . . . 101
7.5 stand-alone scripts . . . 103

8 Files Folders and Configurations 105
8.1 folders . . . 105
8.1.1 program folder . . . 105
8.1.2 user folder . . . 105
8.2 configuration files . . . 105
8.2.1 paths.ini . . . 106
8.2.2 mla_config.ini . . . 106
8.2.3 firmware file . . . 106
8.2.4 calibration files . . . 106
8.2.5 start-up script . . . 106

9 Analog interfaces 107
9.1 signal outputs . . . 107
9.2 signal inputs . . . 107
9.3 aux outputs . . . 108
9.4 Trigger Output . . . 108
9.5 Trigger Input . . . 109
9.6 Clock Reference . . . 109

10 Advanced Hardware and Software Topics 111
10.1 Calibration . . . 111
10.1.1 changing the active calibration . . . 111
10.1.2 creating a new calibration . . . 112
10.1.3 calibration file format . . . 112
10.2 Ethernet communication . . . 112
10.2.1 setting the IP number . . . 112
10.2.2 resetting the IP number . . . 113

10.2.3 Message format . . . 113
10.2.4 Streams . . . 113
10.2.5 Lockin data packet . . . 113
10.3 Direct Memory Access (DMA) . . . 114
10.4 Low-level programmer's model . . . 114

11 References **117**

12 Glossary **119**
12.1 Table of symbols used in this manual . . . 119
12.2 amplitude . . . 119
12.3 base tone . . . 120
12.4 frequency comb . . . 120
12.5 measurement time window . . . 121
12.6 measurement bandwidth . . . 121
12.7 pixel . . . 121
12.8 tones . . . 121
12.9 tuning . . . 122
12.10 units . . . 122
12.11 waveform period . . . 122

Bibliography **123**

Python Module Index **125**

Contents:

CHAPTER

ONE

MULTIFREQUENCY LOCKIN AMPLIFIER

The Multifrequency Lockin Amplifier (MLA™) from **Intermodulation Products** (also known as the Intermodulation Lockin Analyzer, ImLA™), is a unique measurement system which could be described as a synchronous multifrequency waveform generator, digital oscilloscope, and real-time Fourier analyzing instrument. It is constructed from precision, low distortion, analog components, high speed analog-to-digital (ADC) and digital-to-analog (DAC) converters, and a highly parallel logic circuit called a Field Programmable Gate Array (FPGA). Contact **Intermodulation Products** <support@intermod.pro> if you need help adapting the MLA with your experiment.

The MLA can run as many as 32 synchronous Lockins, ideal for frequency-domain multiplexing or simultaneous measurement of multifrequency linear response. But the MLA™ is particularly powerful for measuring nonlinear response in the form of intermodulation (response at the mixing products of multiple drive frequencies) or harmonics (response at integer multiples of a single drive frequency). The MLA™ can process 3 input triggers making possible *triggered lockin measurement*, where amplitude and phase is recorded in relation an external event, while avoiding transient distortions. It can also send three output triggers to initiate external events based on frequency domain response. With two output ports to drive the system under test and 4 input ports to monitor response, all synchronized to the same clock, the MLA™ is a **multi-port, multi-frequency** and **multi-purpose** measurement instrument.

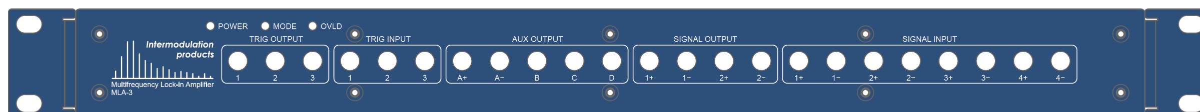
This manual provides the basic information you need to use the MLA™. We start with a brief and general description of the ports. Subsequent sections describe how to use the MLA™ through its *Graphical User Interface (GUI)*. Once you are familiar with the GUI you can start to program the instrument, controlling your measurement through a Python script. Full documentation on all functions needed to program the MLA™ is given in the section on *MLA programming*. For the experts, we have included sections describing *Files Folders and Configurations*, *Calibration*, details about the MLA™ *Analog interfaces*, and a discussion on *Advanced Hardware and Software Topics*.

1.1 connections MLA-3

The input and output ports of the MLA-3 are controlled in the *Graphical User Interface (GUI)* via the *analog panel*.

When you measure (drive) **single-ended**, the voltage is between the center pin and the shield of the SMA connector labeled + (e.g. Input 1+). When you measure (drive) **differential**, the voltage is between center pins of two adjacent SMA connectors labeled + and - (e.g. Input 1+ and Input 1-). A complete description of the input and output ports including schematic diagrams is given in the section on *Analog interfaces*.

1.1.1 front-side



- TRIGGER OUTPUT and TRIGGER INPUT The three input and three output triggers all respond to (send) the same voltage level, which can be set to 2.5 V, 3.3 V or 5.0 V with a jumper on the PC-board. The default factory setting is 5.0V unless otherwise specified when ordering.
- AUX OUTPUT Four slow outputs with a combined speed of 800 kSam/sec. Output A is differential, whereas B,C, and D are single ended. These ports are controlled in the *MLA GUI* via the *aux output panel*.
- OUTPUT Two high speed outputs (500 MSam/sec) with 16 bit resolution. These ports are are always differential. The MLA puts the output voltage between the + port and ground (SMA shield). A voltage of the opposite sign always appears between the - port and ground. If you use these ports differentially, the voltage difference between and + and - ports will be twice the set value. These ports are controlled in the *MLA GUI* via the *analog panel*.
- INPUT Ports 1 and 2 are high-speed (250 MSam/sec) with 14 bit resolution. Ports 3 and 4 have moderate speed (50 MSam/sec) with 16 bit resolution. All ports have switchable range settings, AC or DC coupling, 50 ohm or 1M ohm input impedance. All ports can measure either differential or single-ended signals. When measuring single-ended the - input sees an open circuit (but the - input amplifier is grounded internally). These ports are controlled in the *MLA GUI* via the *analog panel*.

1.1.2 back-side

- REF CLK OUT and REF CLK IN connections are used for synchronizing the internal clock of the MLA™ with the clock of some other piece of measurement equipment. These ports are designed to connect to a 50 Ohm matching impedance. REF CLK OUT sends a 10MHz square waveform, AC-coupled, voltage range 2 Vpp. REF CLK IN must be manually locked to an external reference (see hardware.Hardware.set_clkref_external_10MHz()). Other frequencies and voltage levels can be configured (see *Clock Reference*).
- ETHERNET The MLA™ communicates with the computer via the Ethernet connection. The MLA™ may be directly connected to the Ethernet connection on the computer, or it may be be put behind an Ethernet router, switch or gateway.
- SIGNAL GROUND gives external access to the grounding point for all input signals. This signal ground is isolated from the shielding enclosure (box) of the MLA™.
- POWER GROUND (available on MLA with serial numbers ≥3300) is connected to the enclosure. With the power supply connected, this will also be connected to the power ground (or protective earth) of the house.
- POWER 12 V DC is supplied to the MLA™ via an external power supply. Use only the power supply delivered from **Intermodulation Products**. Using any other supply voids the warranty.

1.1.3 grounding

The MLA-3 has two separate grounds

- SIGNAL GROUND is connected to the outer shield of all SMA connectors.
- POWER GROUND is connected to the enclosure, and via the power supply it is connected to the power ground (or protective earth) in your house.

Both these grounds are exposed as banana connectors on the back side of the MLA-3. The MLA-3 is shipped with a jumper (or busbar) you can use to connect these two grounds.

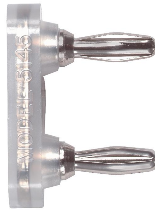


Fig. 1: Jumper (or busbar) shipped with the MLA-3. It can be used to connect the power ground to the signal ground on the back side of the MLA-3.

If the signal ground is left floating, it will not protect against electromagnetic interference. However, if you connected to ground at multiple places, you may get problems with so called *ground loops*. Therefore, as a rule of thumb, signal ground should be connected to power ground at one, but only one, place. So if your device-under-test is for example a passive device with floating ground, you should usually have the jumper connected on the back side of the MLA-3. However, if the MLA-3 is used together with other instruments, these other instruments will often connect the power ground to the signal ground through their power supply. In theses cases, it is usually best to remove the jumper on the back side of the MLA-3 to avoid ground loops.

CHAPTER

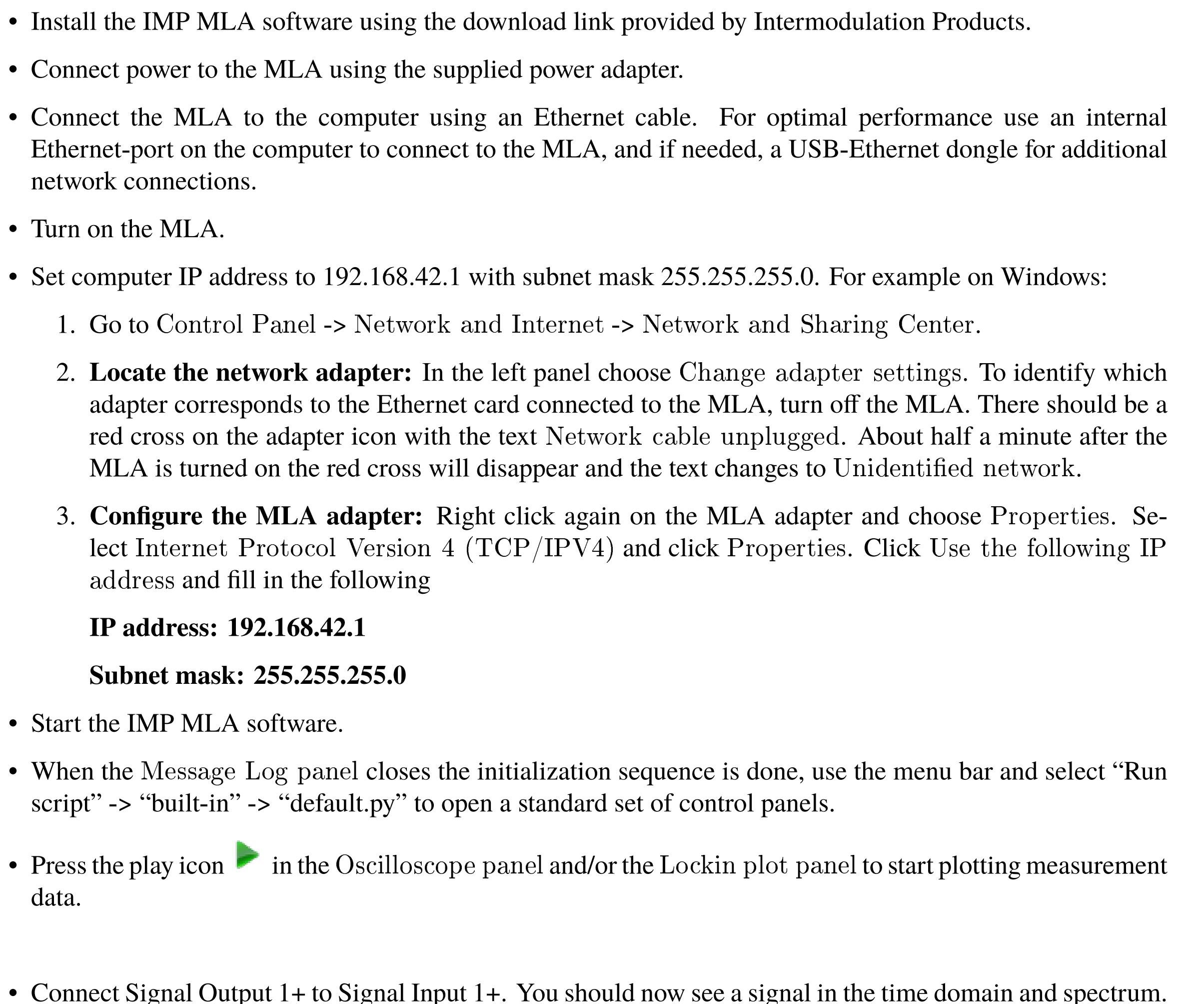
TWO

QUICK START GUIDE

This section describes a quick way to get the MLA and software running.

- Install the IMP MLA software using the download link provided by Intermodulation Products.
- Connect power to the MLA using the supplied power adapter.
- Connect the MLA to the computer using an Ethernet cable. For optimal performance use an internal Ethernet-port on the computer to connect to the MLA, and if needed, a USB-Ethernet dongle for additional network connections.
- Turn on the MLA.
- Set computer IP address to 192.168.42.1 with subnet mask 255.255.255.0. For example on Windows:
    1. Go to Control Panel -> Network and Internet -> Network and Sharing Center.
    2. **Locate the network adapter:** In the left panel choose Change adapter settings. To identify which adapter corresponds to the Ethernet card connected to the MLA, turn off the MLA. There should be a red cross on the adapter icon with the text Network cable unplugged. About half a minute after the MLA is turned on the red cross will disappear and the text changes to Unidentified network.
    3. **Configure the MLA adapter:** Right click again on the MLA adapter and choose Properties. Select Internet Protocol Version 4 (TCP/IPV4) and click Properties. Click Use the following IP address and fill in the following

       IP address: 192.168.42.1

       Subnet mask: 255.255.255.0
- Start the IMP MLA software.
- When the Message Log panel closes the initialization sequence is done, use the menu bar and select "Run script" -> "built-in" -> "default.py" to open a standard set of control panels.
- Press the play icon  in the Oscilloscope panel and/or the Lockin plot panel to start plotting measurement data.
- Connect Signal Output 1+ to Signal Input 1+. You should now see a signal in the time domain and spectrum.

> **Caution:** The instrument is equipped with high precision SMA connectors. They should be handeled carefully for maximum life time. Use only your fingers or the supplied SMA torque wrench when connecting cables.

- Change the amplitude of the first tone in the Lockin setup panel. Press Write to MLA to effect the change.
- Additional functions can be found in the menu bar under "Panels".

CHAPTER

THREE

SOFTWARE INSTALLATION

The MLA™ can be controlled using the dedicated graphical user interface *IMP MLA* software or controlled using the Python API which comes bundled with the software. The software comes in two alternative packages "exe" and "zip". The "exe" version comes bundeled with a Python environment and all the necessary modules and libraries, which make it easy to get started with. It is however limited to work on Windows operating systems and it is not possible to access the Python API from external scripts when using this version. It is also not possible to add additional Python modules for the internal script interface. For those cases the "zip" version, together with a user installed Python, should be used instead.

3.1 "exe" version

To install just double click on the installer file and follow the instructions. After installation is done you will have a link on the desktop called *IMP MLA* which start the software.

Note: You may be prompted with a warning that the installer file is not cryptographically signed with Microsoft. If you made sure you downloaded the file directly from an Intermodulation Products server (intermod.pro or intermodulation-products.com) you can be confident that the software is safe to run in any case.

3.2 "zip" version

This version requires installation of a Python environment on the system. On Windows and Mac OS we recommend Miniconda Python from Anaconda since they provide easy virtual environoment handling. The instructions below assume you are using Miniconda.

- Extract the *IMP MLA* zip-file to a directory of your choice
- Install Miniconda from https://docs.conda.io/en/latest/miniconda.html
- Open an Anaconda Prompt
- **Create a dedicated Python environment for the software** `conda create -n imp python`
  - If needed you can provide specific python version here, e.g. `conda create -n imp python=3.10`
- Activate environment `conda activate imp`
- **Install required modules into the environment** `pip install wxpython numpy scipy matplotlib paramiko configobj`
  - If you are installing the IMP Multifrequency AFM software (*IMP AFM*) some additional packages are needed `pip install h5py scikit-learn`
- Change directory `cd PATH` where PATH is the directory where the zip was extracted to.
- Start software using `python start_mla.py` (on Mac OS you may need `pythonw start_mla.py`)

When starting the software in the future you only need start Anaconda prompt, activate the environment and start the software from the correct path.

CHAPTER

FOUR

MULTIFREQUENCY LOCKIN BASICS

4.1 lockin measurement

The lockin technique is generally defined by its use of a reference oscillation for making a narrow-band measurement of some physical process [Dicke-1946]. The physical system is excited or modulated with a sinusoidal signal of frequency ω, while a phase-locked reference oscillation at the same frequency ω is used to demodulate, or determine the systems response to the excitation.

Let us denote the response signal as $V(t)$. The lockin multiplies this signal with two unity-amplitude reference oscillations and integrates over time, to give two **quadrature components**.

$$I = \frac{1}{T_m}\int_0^{T_m} V(t)\cos(\omega t)dt$$

$$Q = \frac{1}{T_m}\int_0^{T_m} V(t)\sin(\omega t)dt$$

If the response signal is an oscillation at the frequency ω with amplitude A and phase ϕ,

$$V(t) = A\cos(\omega t + \phi)\ .$$

and we perform the integrations over a *measurement time window*, $T_m = M(2\pi/\omega)$, corresponding to an integer number M times the oscillation period $T = 2\pi/\omega$, we find

$$I = \frac{1}{2}A\cos(\phi)$$

$$Q = \frac{1}{2}A\sin(\phi).$$

The quadratures I and Q are the Fourier coefficients of the response, at the reference frequency. Note that at DC, or $\omega = 0$, the integration gives $I = A$ and $Q = 0$. At any non-zero frequency we find the following relation between the quadratures I and Q and the response signal amplitude A.

$$A = 2\sqrt{I^2 + Q^2}$$

The response phase is

$$\phi = \tan^{-1}(Q/I).$$

This response phase is with respect to the phase of the reference oscillation, which has zero phase by definition. It is very useful to represent the response signal as complex number,

$$\hat{V} = I + iQ$$

We call the absolute value of this complex number the **quadrature amplitude**

$$A_{\mathrm{quad}} = \sqrt{I^2 + Q^2} = A/2$$

Note that at DC, or $\omega = 0$, the last equivalence does not hold, and we find $A_{\mathrm{quad}} = A$.

In the time domain the response signal can then be written,

$$V(t) = \hat{V}e^{i\omega t} + \hat{V}^* e^{-i\omega t}$$

where $\hat{V}^* = I - iQ$ is the complex conjugate of $\hat{V}$. Equivalently

$$V(t) = 2\mathbf{Re}\left[\hat{V}e^{i\omega t}\right] = 2I\cos(\omega t) + 2Q\sin(\omega t)$$

4.2 demodulating multiple frequencies

Lockin measurement at one frequency is straight forward, but what happens when there are two frequencies in the excitation waveform and we want to measure the Fourier coefficients of the response at these two frequencies, and perhaps other frequencies. One could use each excitation frequency as a separate reference, but it is far better to define one global reference for all frequencies. The global reference is possible if you carefully choose the excitation frequencies in a process that we call *tuning*, in direct analogy to the tuning of musical instruments. Below we explain why one should tune a multi-frequency lockin measurement, and how tuning is done with the MLA™.

The MLA™ will excite a physical system with a waveform consisting of many different frequency components, and simultaneously measure both quadratures of the response (i.e. demodulate) at many frequencies. We refer to the components of the excitation signal and response signal as *tones*, each having a frequency, *amplitude* and phase (or alternatively frequency, and two quadrature amplitudes). The MLA™ is special in that all tones can be locked to one reference, which is also locked to the *measurement bandwidth* δf, or inverse of the *measurement time window* $\delta T_m = 1/\delta f$. Unlike other multifrequency lockins which are simply many independent lockins in one box (the MLA™ can also run in this mode), the MLA™ uses special digital algorithms to achieve many synchronous lockins, all working off the same reference. This synchronous mode of operation is achieved by *tuning* the frequencies of all tones to ensure that they are integer multiples of δf.

The number of tones N in your MLA™ may depend on the particular firmware that you are running, but typically $N \leq 40$. The amplitude and phase of each drive tone in the multifrequency excitation is set by the user, and each tone can be directed to either or both of the MLA™ output ports. If you do not wish to drive but only 'listen' (measure) a particular frequency, simply set both bits of the output-mask to zero at this tone, or set the output amplitude to zero. The MLA™ measures (demodulates) the quadrature amplitudes of the response at its input ports, at the frequency of each of the N tones. Any tone can be demodulated at one of the 4 input ports.

The demodulation of the response is done by multiplying each digital sample of the input signal with an appropriate demodulating tone and summing over the *measurement time window* T_m, as described above. Thus, T_m plays the role of the time-constant of a traditional lockin. We need a word that refers to the data contained in this time window and we chose the word *pixel*, because the MLA™ was first used in multifrequency AFM [Platz-2008], where each measurement time window corresponds to one image pixel.

4.3 Fourier leakage

When a physical system is driven with a number of pure *tones*, these tones may affect each other in various ways. If the physical system is perfectly linear the principle of superposition applies: The total response is simply a sum, or linear combination of the individual responses at the frequencies of each drive tone. However, Fourier leakage may affect your ability to demodulate and independently measure the various response tones. Fourier leakage occurs when a tone is not perfectly periodic in the measurement time window. A strong tone will 'leak' over in to neighboring frequencies in the spectrum and destroy the measurement of other weaker tones.

A digital lockin samples the signal at discrete times and the quadrature amplitudes are calculated with discrete sums. These sums will be corrupted by Fourier leakage when any of the lockin frequencies and sampling frequency are not commensurate (i.e. do not have a common divisor). Thus, the sampling frequency becomes an ideal reference for all tones in a multifrequency digital lockin. Figure 1 demonstrates the connection between sampling and Fourier leakage.

Fourier leakage can be reduced by so-called windowing, or equivalently, low pass filtering of the demodulated signal, but these methods are imperfect and they can not completely prevent leakage. To completely avoid Fourier leakage, the MLA™ is designed to work only with frequencies that perform an integer number of oscillations in the measurement time window. The process of choosing the best frequencies for measurement is called *tuning*, in analogy to the tuning of musical instruments. The easiest way to tune the MLA™ is to start with a desired measurement bandwidth $\delta f = 1/T_m$ and then choose all frequencies to be an integer multiples of δf.

$$f_i = n_i \delta f \tag{4.1}$$

where n_i are a set of integers.

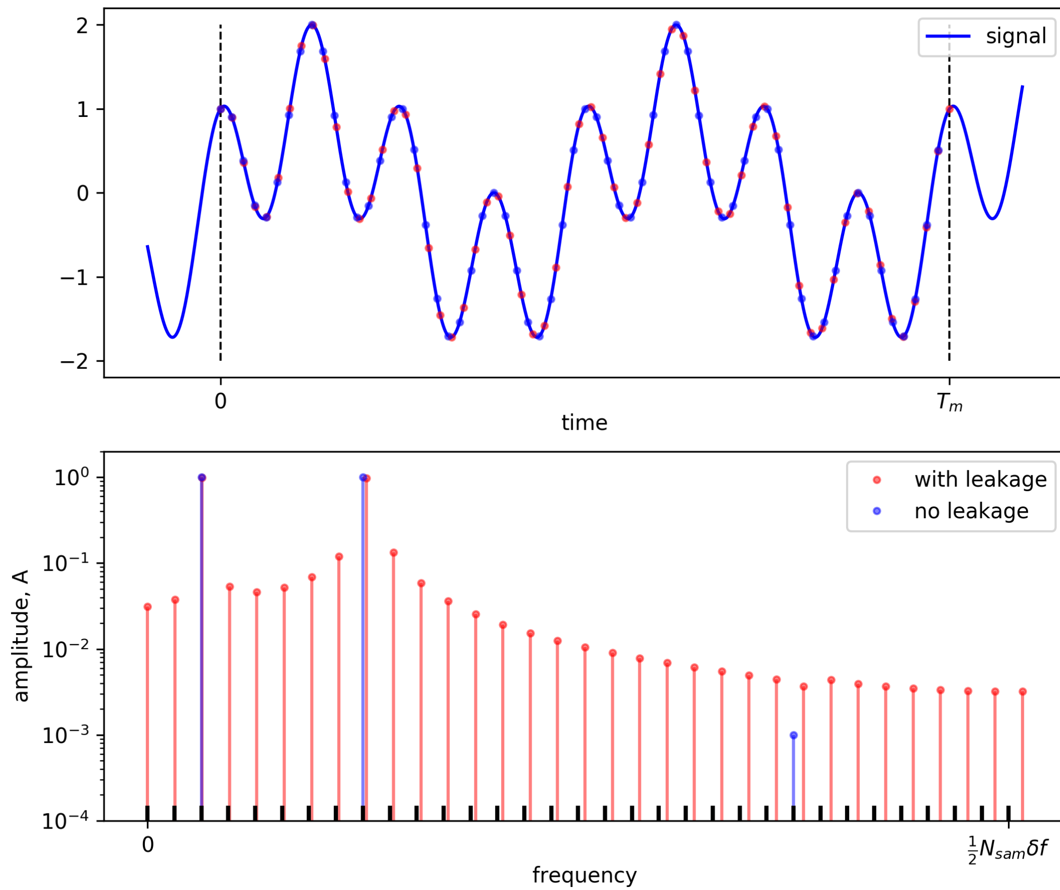


Fig. 1: **Figure 1:** A multifrequency signal with three tones is sampled and Fourier amplitudes are calculated by discrete Fourier sums. The red samples are incommensurate with the frequencies of the tones and the Fourier amplitudes are corrupted with Fourier leakage. The blue samples are commensurate and the spectrum has no leakage, showing amplitude only at the three frequencies contained in the signal. We avoid Fourier leakage by tuning the frequencies of all tones so that they are integer multiples of $\delta f = 1/T_m$, shown by the black grid on the frequency axis.

4.4 tuning and nonlinearity

Fourier leakage is an issue for the measurement of linear systems, but more fascinating is an effect that occurs in nonlinear systems. A nonlinear system will generate new tones in the response waveform that are not in the drive waveform. **Harmonics** are generated at exact integer multiples of the drive frequencies and they will occur in nonlinear systems driven with only one tone. **Intermodulation products**, or mixing products occur when two or more tones are present in the drive, and these occur at frequencies that are integer linear combinations of the drive frequencies. One of the primary design goals of the MLA™ is the accurate measurement of many intermodulation products [Tholen-2011].

When many drive tones are present: $f_0, f_1, f_2...$, nonlinear response is generated at very many new frequencies,

$$f_{\text{IMP}} = k_0 f_0 + k_1 f_1 + k_2 f_2 + ... \tag{4.2}$$

were $k_0, k_1, k_2...$ are any integer.

Equation (4.2) tells us that all frequencies in the nonlinear response will be at integer multiples of a *base tone* Δf, if we choose the frequencies such that they all have a **greatest common divisor** Δf. This property of a tonal system in music is referred to as 'just intonation' or 'just temperament'.

The inverse of the base tone $T = 1/\Delta f$ defines the period of the multifrequency waveform. A positive effect of the constraint (4.1) is that all harmonics and intermodulation products generated by nonlinearity, will also be at frequencies $f_{\text{IMP}} = n\delta f$, where n is an integer. Thus, tuning the measurement also means that the nonlinear response will not be affected by Fourier leakage.

Note that the base tone Δf and the measurement bandwidth δf are not necessarily the same, but they must have an integer relationship,

$$\Delta f = m\delta f \tag{4.3}$$

where $m \geq 1$ is a positive integer. You can improve the signal-to-noise ratio by reducing the measurement bandwidth while keeping the same base tone (i.e. make $m > 1$). Choosing the base tone to be 2, 3, 4,... times the measurement bandwidth allows you to check for period-doubling, period-tripling, period-quadrupeling,etc., bifurcations in nonlinear response.

4.5 setup and tuning functions

When as many as 40 frequencies are involved, setting up the measurement can be a complicated procedure. The MLA™ does have a Graphical User Interface (*MLA GUI*) with fields for adjusting the drive and reading the response amplitude and phase of each individual tone. But setup and measurement in this manner is often not practical. The MLA™ also has a powerful Application Programming Interface (*MLA API*) which allows for full setup and control using commands in a Python script. It is simple to edit these scripts and run them directly in the GUI, making for a very flexible environment to set up and test complex multifrequency measurements.

Within the API there are several convenience functions available to facilitate *tuning*. Tuning functions take a set of target frequencies and a target measurement bandwidth as input arguments. They return a tuned set of frequencies and tuned measurement bandwidth. The set of frequencies can be an entire array of values, or a single 'carrier frequency', around which you might build a *frequency comb*.

The tuning functions behave according to the following rules:

- f, df = mla.lockin.tune0(f, df) performs only the minimum about of tuning necessary to make the requested numbers fit the corresponding digital registers in the MLA™ hardware. No attempts are made to avoid Fourier leakage. This function should be used together with a user-defined tuning algorithm, or in rare cases where Fourier leakage is known not to be an issue, for example when measuring strong tones well separated in frequency. If you are using this function, you will probably want to also use *mla.lockin.reset_awgen_on_new_pixel(False)* to avoid jumps in the signal at the end of each measurement time window (see *finite accuracy and perfect tuning*). See full function documentation in the *MLA API*, subsection *mla.lockin*, lockin.Lockin.tune0().

- f, df = mla.lockin.tune1(f, df, priority='f') gives the standard level of tuning, adequate for most measurements. The returned array of frequencies and bandwidth are selected so that they fulfill the condition (4.1) which avoids Fourier leakage. The key-word argument priority='f' puts the highest priority on getting the frequencies as close as possible to the requested frequencies. With priority priority='df' the highest priority is put on getting the measurement bandwidth as close as possible to the requested measurement bandwidth. See full function documentation in the *MLA API*, subsection *mla.lockin*, lockin.Lockin.tune1().
- f, df = mla.lockin.tune2(f, df, priority='f') results in the most rigid tuning which forces the number of samples in the measurement time window to be a power of 2 (see *finite accuracy and perfect tuning*). In this case the command mla.lockin.reset_awgen_on_new_pixel(False) has no effect because there is no phase correction. This tuning is normally not used because it significantly restricts the available frequencies. See full function documentation in the *MLA API*, subsection *mla.lockin*, lockin.Lockin.tune2().

The MLA™ is not configured by simply calling a tuning function. The separate functions mla.lockin.set_frequencies(f) and mla.lockin.set_measurement_bandwidth(df) transfer the tuned values to the MLA™, configuring it for measurement. If these functions are used without tuning, the measurement will suffer from Fourier leakage and phase drift. It is therefore up to the user ensure that the condition (4.1) is fulfilled, either with the above tuning functions, or with their own tuning algorithm.

For further discussion of tuning algorithms, see *finite accuracy and perfect tuning*.

4.6 tuning frequency combs

The three tuning functions described above do not represent the only possible ways to tune the measurement. As with tuning systems in music, there many ways to choose the tones and the best way may depend on the particular measurement that you would like to perform. You can experiment with your own tuning methods, but to avoid phase drift and Fourier leakage, one should conform to the constraints (4.1) and (4.3) above. In this case the frequencies of all tones will be defined by set of integers n_i and a measurement bandwidth δf.

We call waveforms of this type a *frequency comb*, because all tones exist on a equally spaced grid of frequencies, like teeth of the comb for your hair. You can choose to place your N measurement and excitation frequencies where ever you want on this grid. Once you have constructed the set of integers n_i, use the following function to configure the MLA™:

- ns, df, f = set_frequencies_by_n_and_df(n, df, tune=False, wait_for_effect=True) takes and array of integers n_i and a bandwidth δf and sets the lockin frequencies to be $f_i = n_i \delta f$. The function returns the number of samples per *pixel* n_s, the tuned bandwidth δf and an array with the set frequencies f_i. See full function documentation in the *MLA API*, subsection *mla.lockin*, lockin.Lockin.set_frequencies_by_n_and_df().

The array of integers n_i is often calculated relative to some tuned ‘carrier’ frequency. It is best to call this function using a tuned bandwidth δf and carrier frequency, both of which are first determined using one of tuning functions described above. We demonstrate this in the following example script:

```
# Example for setting a comb of 5 pairs of tones equally spaced around a single carrier frequency
# Choose the carrier frequency and measurement bandwidth
_fc = 100e3  # Target carrier frequency
_df = 1e3  # Target measurement bandwidth
nr_freq = 11  # Number of frequencies in the comb

# Tune fc and df
fc, df = mla.lockin.tune2(_fc, _df)  # perfect tuning, priority on carrier frequency

# Build the frequency comb
n_c = int(np.round(fp/df)) #integer corresponding to the tuned carrier frequency
n_low = n_c - np.arange(nr_freq/2, 0, -1)
n_high = n_c + np.arange(1, nr_freq/2+1, 1)
n_comb = np.concatenate(([n_c], n_low, n_high))
```

(continues on next page)

(continued from previous page)

```
# Configure lockin with the comb of frequencies
mla.lockin.set_frequencies_by_n_and_df(n_comb, df)
```

For further discussion of tuning algorithms, see *finite accuracy and perfect tuning*.

4.7 advanced lockin topics

The general introduction above is augmented by several advanced sections with a more detailed explanation of the particular features and concepts behind the MLA™. If you feel that an important explanation is missing from this manual, please contact **Intermodulation Products** <info@intermodulation-products.com> with questions and suggestions. We appreciate the feedback!

4.7.1 finite accuracy and perfect tuning

Due to the finite precision of numeric representation, and due to algorithm-specific limitations, there are many subtleties to the creation of 'perfect' tuning when using digital electronics and discrete math. Each measurement time window T_m is ultimately defined by a number of samples performed by the AD-converter $n_s = f_s T_m$, where the **sampling frequency** f_s is is fixed by the MLA™ internal clock (but can be adjusted within limits by a software command). The MLA™ will generate pure tones at the 'exact' frequencies $f_i = k_i \delta f$, only when n_s is a power of 2. In other words, perfect tuning is achieved only when the measurement bandwidth δf and the sampling frequency f_s satisfy $\delta f = f_s / 2^n$, where n is an integer. If this condition is fulfilled we call it 'perfect' tuning because the algorithms for modulation and demodulation are not a source of error, and the theoretical error is at the limit determined by the finite precision of the digital numbers used in calculating the Fourier sums. This theoretical limit is almost always very far below the noise level, or experimental error in an actual measurement.

Perfect tuning is achieved with the function `lockin.Lockin.tune2()`, which enforces the integer-power-of-two restriction on the number of samples in the time window. Unfortunately, this restriction also has the effect of significantly limiting the number of possible frequencies available for modulation and demodulation. Therefore, other algorithms have been developed that achieve very close to perfect tuning and in nearly all experimental situations they are indistinguishable from perfect tuning. The user can apply the function `lockin.Lockin.tune1()`, or devise their own tuning algorithm that maintains the constraint on the bandwidth and frequencies ($f_i = n_i \delta f$, n_i a set of integers) and apply the function `lockin.Lockin.set_frequencies_by_n_and_df()`. Note that when using the later, it is best to first tune the bandwidth δf with the help of a tuning function (see *tuning frequency combs*).

When the tuning is imperfect it is desirable to apply a correction at the end of each time window which resets the phase counter, to avoid a very slow drift of the phase. This reset will not be noticeable since it corresponds to a tiny correction of phase between each measurement time window. In other words, the theoretical error resulting from this reset is significantly lower than the measurement noise for any reasonable measurement bandwidth. Nevertheless, if for whatever reason you need absolute phase-stablity of a drive tone over a long time, this reset action can be turned off by the function `lockin.Lockin.reset_tones_on_new_pixel()` with the argument `False`.

When the phase reset is turned off and a measurement is running for a long time, a slow drift in the demodulated phase may become noticeable in some circumstances: The output tone and the demodulation tone drift equally, so the drift will not be noticeable in the measured response of a linear system. However when measuring nonlinear response in the form of intermodulation between two drive frequencies, a slow phase drift can be observed when the phase reset is turned off. Another instance where this phase drift can become noticeable are measurements that use signals supplied by external devices, even when these devices are locked to the MLA™ clock. In this latter case it is recommended that you use the most rigid tuning function `lockin.Lockin.tune2()` and turn off the reset action with `reset_tones_on_new_pixel(False)` .

4.7.2 frequency sweep

The MLA™ has built in functions for performing frequency sweeps. Sweeps can be done in the standard way, with only one frequency applied during each measurement time window. But the MLA™ also has a unique and powerful feature which allows for multiple frequencies to be applied in each time window. Using this multi-tone frequency sweep will dramatically reduce the time needed to sweep over a band at some given frequency resolution, and it can be very useful if you are searching for a very narrow spectral line in a broad band. However, the multi-tone sweep will not give the same result as the standard single-frequency method, if the system under test is nonlinear. The nonlinear system will experience intermodulation between the multiple applied tones, making the principle of superposition no longer valid.

Frequency sweeps are initiated with one command, and the data is read from the MLA™ with a subsequent command. For full documentation of the sweeping functions see the *MLA API*, subsection *mla.lockin*, `lockin.Lockin.frequency_sweep()` and `lockin.Lockin.get_frequency_sweep()` and for multifrequency sweeping see `lockin.Lockin.frequency_sweep_multifreq()` and `lockin.Lockin.get_frequency_sweep_multifreq()` .

4.7.3 triggered lockin measurement

The MLA™ has the ability to trigger lockin measurement. This feature was originally developed to synchronize data acquisition with a the scan of a scanning probe microscope, but it is useful for many other measurements as well. When the MLA™ receives a trigger event, it can set up so that all the phase counters jump to their initial positions (determined by the phase setting of all tones). When this reset happens in the middle of a time window, there will be a glitch in the drive waveform, resulting in a transient in the system under test. This transient may be undesirable, so it is possible to turn off the reset action with `lockin.Lockin.reset_tones_on_trig1()`. With this reset action turned off, the receipt of a trigger does not effect the measurement at all.

Receipt of a trigger also increments a trigger counter. There are two trigger counters, one for each of the two trigger ports `TRIG IN 1` and `TRIG IN 2` . These counters are transmitted with every pixel of lockin data in the data stream (see *Lockin data packet*). When triggered lockin measurement is enabled with `lockin.Lockin.enable_wait_for_trigger()` the MLA watches the trigger counter. The `lockin.Lockin.wait_for_trigger()` function will block the computer thread until the counter is incremented, and then return with the buffer position of the lockin data packet, during which the trigger was received. The user can then read out data relative to this buffer position, either pre-trigger data or post-trigger data. In scanning applications, `TRIG IN 1` is used for the End Of Line (EOL) and `TRIG IN 2` is used for End of Frame (EOF).

The triggered measurement mode described above is included with a standard shipment of the MLA™. However, with 6 trigger ports and the ability to do real-time logic with the FPGA firmware, there are endless possibilities to do complex triggering by modification of the MLA™ firmware. Contact **Intermodulation Products** <info@intermodulation-products.com> if you have special triggering needs.

4.7.4 real-time feedback

Any real-time reaction to the measured data must be low level-programmed in the MLA™. One example of this is the feedback used in multifrequency atomic force microscopy (AFM): The MLA™ is programed to deliver a frequently-updated voltage to an output port, which is proportional to the response amplitude of one tone. The currently available feedback functions are documented in *mla.feeback*. There are an infinite number of possible feedback ideas that can be realized by low-level programming of the MLA™ firmware. Consult with **Intermodulation Products** <info@intermodulation-products.com> if your measurement requires custom feedback.

4.7.5 demodulation details

To optimize the usage of resources for calculating the quadrature response, the internally-generated tones are used both to synthesize the drive waveform, and to demodulate the signal at the input port. Changing the amplitude of a drive tone does not affect demodulation, because demodulation multiplies with a unity-amplitude tone and sums to find the Fourier coefficients. However, when the phase of an internally-generated tone is changed, both the output and the demodulating tone will change phase by the same amount. Thus the response phase (with respect to the drive phase) actually does not change. However, because all tones are generated from the same digital clock, the MLA™ can keep track of the absolute phase of each tone, and rotate the demodulated quadratures so that they are presented with 'absolute' phase, or phase with respect to the phase of one reference, the *base tone*.

If, for what ever reason, you need to have a demodulation phase which is independent of the drive phase, this can easily be done, but it requires that you sacrifice one frequency for analysis. Simply set two of the tones to the same frequency, configuring the drive at one tone to be zero (i.e. measurement only). Some versions of the firmware are designed to work with a different number tones for modulation and demodulation. Consult with **Intermodulation Products** <info@intermodulation-products.com> if you have special needs in this regard.

CHAPTER

FIVE

GRAPHICAL USER INTERFACE (GUI)

With simple mouse clicks from the MLA™ GUI you can setup and control the instrument, record measurement data, plot data and save data. For many measurements working only with the GUI is sufficient. More complex measurements are programed with Python scripts and these can be run inside the GUI from the `script panel`, as described in the section on *MLA programming*.

The GUI consists of a set of **panels**, each with different functionality. All panels are opened from the `Panels` pull-down menu in the top menu bar. Closing a panel removes it from view, but it does not kill the instance of the panel in the software, and it may be re-opened in the same state that it was closed. Panels can be free-floating and re-sized, or docked in the main frame with different arrangements. Below we give an overview of the functionality and controls of each panel.

5.1 common features

All plots have a toolbar and many toolbars have the following common icon buttons.

Clear plot : Press to clear the plot.

Pan-Zoom: when selected, a left-click-and-drag will cause the plot to zoom, starting from the point of click. Dragging horizontally will zoom only the x-axis, dragging vertically will zoom only the y-axis. Dragging at an angle controls the relative rate of zoom of each axis. A right-click-and-drag will grab the plot at the point of click and slide it in the direction of the drag. Performing these actions while holding down the **x**, **y** or **ctrl** keys will restrict the pan or zoom to occur only in the x-axis, y-axis, or preserving current aspect ratio, respectively.

Zoom : zooms to a selected rectangle with a right-click-and-drag over the plot.

Toggle visibility : Use this tool to individually toggle the visibility of each line trace in the plot.

Open saved line trace : Open and plot saved line traces from text files saved with the Save data tool.

Save data : Save the data in the plot to a text file.

Save image : The Save image button opens a dialog box for saving the figure to file. Several different formats (png, eps, pdf and more) are available. Sometimes you would like to save plots with a different aspect ratio, or change the relative size of the frame and text. Simply rescale the entire GUI or free-floating panel (click-and-drag on the lower right corner) before you save. This action will rescale the figure keeping plot limits, text and line size fixed.

Start measurement : Press to start a measurement, depress to stop.

Mouse actions on many plots allow for quick rescaling.

- Hover over a plot and use the mouse wheel to zoom the y-axis from the pointers y location.

- Hover over the x-axis just outside the plot and use the mouse wheel to zoom the x-axis from the pointers location.
- Center-click either x-axis or y-axis just outside the plot frame to scale with the value in Axis settings.
- Left-click either x-axis or y-axis just outside the plot to rescale that axis to include all data.
- On some plots, a right-click anywhere inside the plot will open the Axis settings panel where you can enter the axis limits.

5.2 analog panel

The Analog panel controls the configuration of the MLA input and output ports. Here you can set the range, coupling (AC or DC) and the type of signal to be measured (differential or single-ended). The panel has one tab for each input and output port on the MLA-3. This panel has no function with the generation 2 MLA.

IN 1 to IN 4

There are two modes for setting up the inputs:

Configuration mode has an interface very similar to a standard oscilloscope. Choose between:

- DC or AC coupling.
- single-ended or differential measurement (see *connections MLA-3*).
- 50 ohm or 1M ohm input impedance.
- select an input Range, from ± 40 mV to ± 10 V full scale.

Manual switch control When activated, this mode allows for manual routing of the signal in the analog input stage. Use this mode if you require a special input configuration. Refer to the schematic diagrams in the section *signal inputs*. The switches can be set from the *MLA API* with the function hardware.Hardware.set_input_relay(). When Configuration mode is activated, the state of each switch is displayed (checked = True).

> **Caution:** When both switches dc=True and bypass=True, both + and - inputs will be pulled up to 1 V with respect to the grounded shield.

OUT 1 and OUT 2

- Select the output Range, either ± 2 V to ± 12 V full scale.
- Note that OUT 1 and OUT 2 ports are always differential. The MLA puts the output voltage between the + port and ground (SMA shield). A voltage of the opposite sign always appears between the - port and ground. If you use these ports differentially, the voltage difference between and + and - ports will be twice the set value.

> **Caution:** The 12 V range is rather powerful and you can burn stuff with it. Be careful with what you are driving when using the 12 V range on these ports.

5.3 aux output panel

OUT A - OUT D

Manually control the DC Bias for the **AUX OUTPUT** ports. For OUT A you can select the option Tone 0 amplitude to output a voltage proportional to the amplitude of tone 0. Change the Gain factor to change the the proportionality constant. You can also add and Offset to the output voltage. This feature is useful if you would like to do amplitude feedback, or example in AFM. Other types of feedback are possible, as documented in feedback.Feedback.

5.4 lockin setup panel

The lockin setup panel can manually configure each tone in the MLA™, or view the configuration made by a script. The information in all fields in this panel is not updated automatically, so the buttons at the bottom are important:

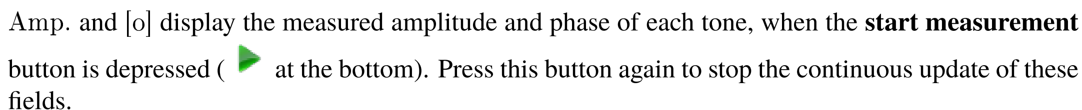
> Read from MLA will read the present configuration and display it in the panel. You can then edit any white-background field in the panel.
>
> Write to MLA sets the MLA™ to the present state of the panel. Press this after editing to effect the change.
>
> Tune...prio... click on this selector to choose which tuning function will be applied when Write to MLA is pressed. See *setup and tuning functions* for a description of the tuning functions.

At the top of the panel:

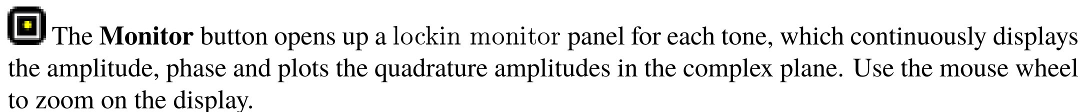
> Measurement time window [ms] T_m and Measurement bandwidth [Hz] δf are not independent. $\delta f = 1/T_m$. Changing one will force the other to change.
>
> Load and Save As buttons allow you to store configurations and reload them.

There is one row for each tone in the MLA™. The columns are:

> Freq. [kHz] shows the frequency of each tone. You can select n [-] to display (control) the frequency as an integer multiple of the Measurement bandwidth [Hz].
>
> Amp shows the amplitude of each tone. You can select to display (control) the amplitude in Volts [V] , or percent of full amplitude [%].
>
> Phase [o] displays (controls) the phase in degrees.
>
> Out 1 and Out 2 check boxes directs each tone to the output ports. When checked, the tone is applied to that port.
>
> IN Port displays (controls) at which input port the MLA™ will listen for each tone. You can choose between input ports 1 to 4, only one input port for each tone .



> Amp. and [o] display the measured amplitude and phase of each tone, when the **start measurement** button is depressed (at the bottom). Press this button again to stop the continuous update of these fields.

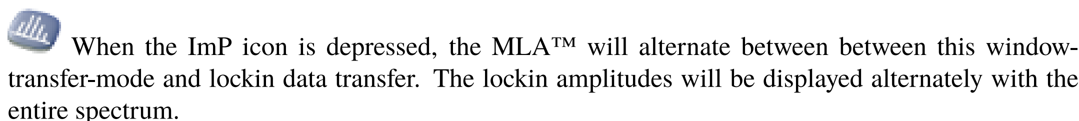


> The **Monitor** button opens up a lockin monitor panel for each tone, which continuously displays the amplitude, phase and plots the quadrature amplitudes in the complex plane. Use the mouse wheel to zoom on the display.

5.5 oscilloscope panel

The oscilloscope panel runs the MLA™ in a mode where all samples, or a down-sampled fraction of samples in one measurement time window, are transferred to the computer. The Fast Fourier Transform (FFT) of the data is taken in the computer and the amplitude is plotted vs. frequency, revealing the *entire* response spectrum.

Start and Stop visual updates with the Start measurement button.



When the ImP icon is depressed, the MLA™ will alternate between between this window-transfer-mode and lockin data transfer. The lockin amplitudes will be displayed alternately with the entire spectrum.

Selects the channels to display with the numbered buttons.

Trig Opens a selector to control how the measurement is triggered.

Free - Auto-triggers to get new data when plot is finished.

Pixel - Auto-triggers at the start of the next lockin *measurement time window*.

Pixel and TRIG IN 1 - Triggers at the start of next measurement_time_window, when a trigger is received at Trigger Input 1.

Toggels open/close data fields above the tool bar to set:

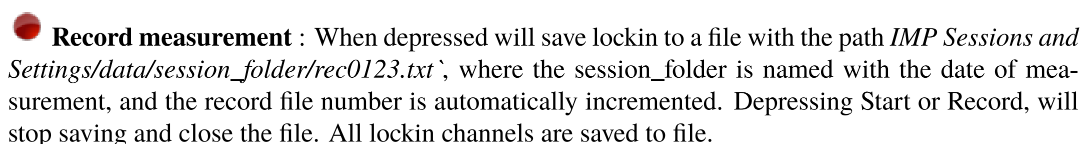
Number of samples: Number of samples in the time window displayed, and analyzed by FFT.

Downsample factor: will average the given number of samples before transfer to the computer. Down-sampling reduces the amount of data handled, resulting in faster Fourier analysis and plotting, but it also reduces the time resolution and the maximum frequency in the Fourier transform.

Use value from Lockin when checked, sets the number of samples to that used by the lockin.

5.6 lockin history panel

The Lockin history shows the most recent lockin data plotted as a function of time.



Record measurement : When depressed will save lockin to a file with the path *IMP Sessions and Settings/data/session_folder/rec0123.txt`*, where the session_folder is named with the date of measurement, and the record file number is automatically incremented. Depressing Start or Record, will stop saving and close the file. All lockin channels are saved to file.

The history can be displayed in a multitude of ways:

Set number of subplots: controls the matrix of subplots in the figure.

Right-click on each subplot to open up the Axis settings dialog box for that subplot.

Select frequencies -> opens a dialog with check boxes to select the frequency (with its associated input port) to be plotted in this subplot.

Quantity: selects which lockin data to plot: Amplitude, phase, I-quadrature, or Q-quadrature.

Scale type: selects the scale of the vertical axis: Linear, Logarithmic or Modulo where the latter will wrap between Y min and Y max .

Legend: controls the placement of the legend in the subplot axis.

Y min, Y max, history length [s] control the subplot limits. These limits are applied hitting enter on the keyboard, or clicking on the plot.

5.7 script panel

The MLA™ GUI has a built-in script editor with the standard functionality, including the ability to execute the script with the Run button. It is often not practical to have buttons and knobs to individually control each tone. Furthermore, setting up a multifreqeuncy measurement requires some thought and calculations are needed to make sure that all tones fit in to the same 'key' (see *setup and tuning functions*). Configuring the measurement is best done with a script.

The Startup button indicates the script to be opened and executed when the software is started. Press this button to change the startup script to the currently open script. The system is delivered with a default.py startup script stored in IMP Sessions and Settings/settings/mla_scripts/built-in. This script can be modified and saved with a new name in the IMP Sessions and Settings/settings/mla_scripts folder. At every start of the MLA™ software, all scripts in the built_in sub-folder will be overwritten and restored to their originally delivered version.

A more complete discussion of **Python** scripts and detailed documentation of all functions that control the that control the MLA™ is given in the chapter on *MLA programming*, complete with *example scripts*.

5.8 script plot panel

The script plot panel is a panel with an empty figure that the user can program to plot data in a script. See *MLA programming* for a discussion of how to write and run scripts. There you will find an example script for *plotting in the GUI*

5.9 frequency sweep panel

One often measures response at very many frequencies spread evenly over some band, for example when searching for a very sharp resonance. The frequency sweep panel is designed for this purpose. The sweep is actually not continuous, but a sequential stepping between closely spaced, discrete frequencies. This type of measurement is speed up significantly when using the multifrequency capability of the MLA™.

> The Run sweep button starts the measurement, changing to Stop when activated.
>
> Edit the Start frequency [kHz] and Stop frequency [kHz] fields to change the sweep range.
>
> Bandwidth sets the *measurement bandwidth*, or the inverse of the *measurement time window*, which is time needed to measure the response.
>
> Number of points sets the total number of points in the sweep range. Note that the frequency spacing between points will not be exactly equidistant, but as close as possible while maintaining the constraints on bandwidth and drive frequencies discussed in the section on *Fourier leakage*.
>
> Drive amplitude [V] controls the peak amplitude at the output port.
>
> Use multifrequency when activated, will use all available lockins to collect many points in parallel. This speeds up the measurement considerably, while keeping the bandwidth and therefore SNR at the same level. With multifrequency a sweep will only give the same result as a single-frequency sweep if the system is linear. A nonlinear system will induce intermodulation between tones in the multifrequency drive.
>
> Locate Peak when selected will automatically zoom on the maximum value in the sweep and perform a more detailed sweep. This feature is very useful for resonance finding. When it is activated, the Resonance frequency and Quality Factor are determined by fitting to a Lorentzian line-shape and displayed at the bottom.
>
> Use Correction allows one to 'de-embed' linear response from other spurious response in measurement chain between input and output of the MLA™. Sometimes electronic equipment between the MLA™ ports and the device-under-test (DUT) can cause frequency-dependent amplitude and phase

shifts to the signal. Use Correction compensates for any linear transfer of the signal due to these components 'embedded' in the measurement chain. The compensation is made using a procedure called de-embedding, performed in two steps:

1. A frequency sweep is performed while by-passing the DUT (insert a short in place of the DUT). After this sweep, press the Set Correction button, which transfers the result of this sweep to an internal correction file.
2. Put the DUT back in to the measurement chain and check the Use Correction check-box. All subsequent sweeps performed with this box checked will correct for the amplitude and phase shifts measured in the first step.

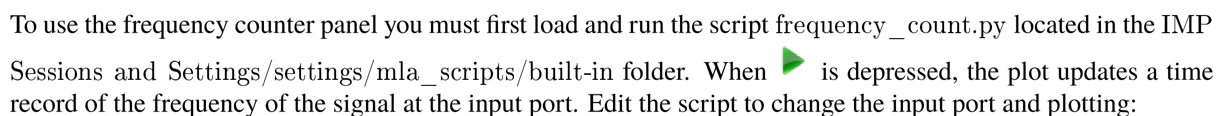
Note: you must do this procedure again if you change anything in the measurement chain, or change any settings that configure the sweep.

In summary:

- Use Correction will correct the measurement for an arbitrary *linear*, frequency-dependant transfer function embedded between the output and the input points of the measurement chain.
- Set Correction stores the last-measured sweep to an internal file and uses this file for the correction.

Clear Plot removes all previous sweeps and the fit.

5.10 frequency counter panel

To use the frequency counter panel you must first load and run the script frequency_count.py located in the IMP Sessions and Settings/settings/mla_scripts/built-in folder. When  is depressed, the plot updates a time record of the frequency of the signal at the input port. Edit the script to change the input port and plotting:

samples_per_measurement defines the time window used to determine the frequency. Must be a power of 2.

downsample_factor is the number of raw ADC samples averaged, to create one measurement sample.

frequency_correction_factor software calibration constant.

port controls the MLA™ for the frequency counter function.

5.11 message log panel

The message log panel contains a record of output from the software. It is very useful debugging problems with scripts. Print statements in a script will be output here.

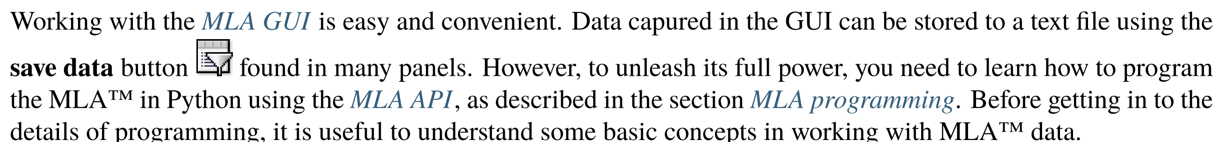
5.12 Python shell panel

The Python shell panel allow you to execute Python statements in real time. All functions controlling the *MLA GUI* and the *MLA API* are available in this shell. The shell shares its namespace with scripts executed in the script panel, so that any variable or function created in a script can be accessed from the shell and vica-versa.

CHAPTER

SIX

COMMUNICATING WITH THE MLA™

Working with the *MLA GUI* is easy and convenient. Data capured in the GUI can be stored to a text file using the **save data** button  found in many panels. However, to unleash its full power, you need to learn how to program the MLA™ in Python using the *MLA API*, as described in the section *MLA programming*. Before getting in to the details of programming, it is useful to understand some basic concepts in working with MLA™ data.

6.1 data streams

The MLA™ is designed for fast and continuous sampling of a signal. When used with the *setup and tuning functions*, the signal is optimally sampled for discrete Fourier analysis. Discrete Fourier analysis on the optimally sampled time stream can then be performed on the computer using the Fast Fourier Transform (FFT). The result is the full Fourier spectrum. However, time streams can not go on indefinitely at the full speed of the DAC, as this generates is too much data too fast. Lengthy times streams require down-sampling, or some other form of compression, for example the lockin (Foruier analysis).

The MLA™ lockins perform the Fourier analysis in real-time on the FPGA on a limited number of user-defined frequencies. Immediately after the acquisition of last sample in a *measurement time window* T_m, the MLA™ starts then next lockin calculation, without skipping a single sample. The lockin data is transfered to the computer as a **lockin stream**, representing a partial spectogram of quadrature amplitudes evolving on a time scale slower than T_m, with no spurious phase jumps.

The streams of lockin data or time data are buffered in different stages, the first being one Gigabyte memory inside the MLA™. This MLA™ buffer is continuously filling in new data and emptying old data. When the command lockin.Lockin.start_lockin() or osc.Osc.start_time_stream() is issued, an automatic data transfer will begin, sending the data down stream to the computer via Ethernet. The Ethernet receiver runs as a process in the computers operating system which is separated from the rest of the measurement software. This separation ensured that the computer is always ready to receive new data, even when processing or plotting the data is blocking the measurement software.

Below we describe how to work with data streams in general terms. For detailed documentation and additional example scripts, refer to the section on *MLA programming* and the documentation of the *MLA API*.

6.2 streaming lockin data

Lockin data for all *tones* calculated during one *measurement time window* forms a unit of data that we call a *pixel* . This pixel is bunched together with some meta data to form a *Lockin data packet*. The command lockin.Lockin.start_lockin() starts a stream of lockin data packets to the computer. During this stream, multiple readout commands are issued to extract buffered data and bring it in to Python. The extraction commands are either lockin.Lockin.get_pixels() or lockin.Lockin.get_new_pixels().

This code example waits for one pixel to finish, then gets that pixel.

```
mla.lockin.start_lockin()
mla.lockin.wait_for_new_pixels(1)
pixels, meta = mla.lockin.get_pixels(1)
mla.lockin.stop_lockin()
```

Typically the user will have a loop to read data and perform other tasks such as plotting and further processing. While the computer CPU is loaded with these tasks, new measurement data will be buffered for the user to extract in the next iteration of the loop. See the example script *lockin measurement loop*. Refer to the Python documentation `lockin.Lockin` for full details.

6.2.1 lockin data formats

Multifrequency lockin data is contained in numpy arrays with different formats, depending on how the data is used. Implicit in these formats is the mapping of the array index to a *tones* in the MLA, which we call the **tone index**. The mapping between the tone index and the actual frequency of the tone is done with the **frequency array**:

freqs = np.array([$f_0, f_1, f_2, \ldots, f_{N-1}$])

where N is the number of tones in the MLA™. Note that in this context, f_0 is not the resonance frequency, but the frequency of the tone with index 0. The frequencies can be expressed in Hz, for example when using `lockin.Lockin.set_frequencies()`. Before using this function, tuning should be performed (see *setup and tuning functions*). Alternatively you can specify the frequencies as an array of integers, which are the multiples of the measurement bandwidth, and use `lockin.Lockin.set_frequencies_by_n_and_df()`.

The response at all measured frequencies, referred to as a *pixel* of lockin data, are expressed in different ways that are all equivalent and can be transformed to one another. Depending on how this data is used, you will encounter the following formats:

6.2.1.1 IQ-real

An array of real numbers, with the quadrature amplitudes at each tone listed pair-wise.

- pixel = np.array([$I_0, Q_0, I_1, Q_1, I_2, Q_2, \ldots, I_{N-1}, Q_{N-1}$], dtype=float)

This format follows the *Lockin data packet*. Note that the array size is 2N, and the index mapping here must step by two in order for the tone index of the I and Q values to coincide with the tone index of the frequency array.

6.2.1.2 IQ-complex

An array of complex numbers, the real part being the amplitude of the I quadrature and the imaginary part the amplitude of the Q quadrature.

- pixel = np.array([$I_0 + iQ_0, I_1 + iQ_1, I_2 + iQ_2, \ldots, I_{N-1} + iQ_{N-1}$], dtype=complex)

This format is useful when performing Fourier analysis, as numpy FFT functions will work on complex arrays. Note that the array size is N, so the tone index here does coincide with the tone index of the frequency array.

6.2.1.3 amp and phase

Two arrays of real numbers

- pixel_amp = np.array([$\sqrt{I_0^2 + Q_0^2}, \sqrt{I_1^2 + Q_1^2}, \sqrt{I_2^2 + Q_2^2}, \ldots, \sqrt{I_{N-1}^2 + Q_{N-1}^2}$], dtype=float)
- pixel_phase = np.array([$\tan^{-1}(I_0/Q_0), \tan^{-1}(I_1/Q_1), \tan^{-1}(I_2/Q_2), \ldots, \tan^{-1}(I_{N-1}/Q_{N-1})$], dtype=float)

Here again the array size is N, so the tone index does coincide with the tone index of the frequency array.

6.2.1.4 data conversion

Depending on what you are doing, different formats may be required. For example when setting drive frequencies, the MLA™ set functions want amplitude and phase as input arguments. Some get functions allow you to get the data in different formats. Whatever the case, it is easy to convert between the different formats using array operations in numpy.

6.3 streaming time data

Like lockin streams, time streams also need to be started and stopped. However time steams differ in that Python is always blocked until the acquisition is finished. The function osc.Osc.get_data() will wait until the specified number of samples is acquired.

```
mla.osc.start_time_stream(50000)
mla.osc.get_data(50000)
mla.osc.stop_time_stream()
```

Time data is currently only available as integer ADU's. There are several methods for optimally down-sampling the data for faster FFT analysis on the computer. Refer to the Python documentation osc.Osc for more details, or contact **Intermodulation Products** <info@intermodulation-products.com> if you need further help acquiring time streams.

CHAPTER

SEVEN

MLA PROGRAMMING

The MLA™ comes with a Python software interface consisting of two main parts:

- *MLA API* (Application Programming Interface) is a software library for accessing the MLA™ hardware, set up and control of measurements with the MLA™, and reading out the measurement data from the MLA™.
- *MLA GUI* (Graphical User Interface) is a versatile interface for setting up and performing many different types of measurements with the MLA, plotting the results, and saving the data to the computer for further analysis.

Thanks to Python's interactive environment it is possible to use the MLA API directly from the Python command line. Command line statements can be bunched together into scripts for automated set up and measurement. Such scripts could then be run stand-alone, integrated into other software, or run from the script panel built in to the MLA GUI. The same API is used in all cases and it serves as the foundation for full-featured interactive graphical applications, such as the MLA GUI.

The MLA GUI can be used entirely without programming, but many tasks are still best performed by scripting. Therefore, a *Python shell panel* for running single Python commands, and a *Script panel* for writing and executing Python scripts, is available inside the MLA GUI. The combination of a GUI and programming capability in the same application can be very practical in many situations. For example, you can use the *Oscilloscope panel* and the *Lockin plot panel* for debugging your measurement setup. When all looks fine, you can start your script which performs the measurement, data analysis and plotting. The MLA GUI includes a rich set of built-in scripts that can be used as templates for your custom scripts.

There are many options for programing the MLA™, but no matter which option you choose, you use the **same** *MLA API*, so that all MLA™ commands look exactly the same. You can test your commands in the *Python shell panel* and the move them over to the *Script panel*. When you are building a larger a stand-alone measurement script, you use the *Script panel* in the MLA-GUI to test and debug pieces of code, and then paste the code into your stand-alone script.

7.1 NumPy and matplotlib

The MLA software uses the open-source libraries NumPy for array math and matplotlib for plotting. Together, NumPy and matplotlib provide an experience much similar to the well-known MATLAB © software. Both NumPy and matplotlib are well documented on their respective web pages. In particular the matplotlib gallery is very useful. If you are familiar with MATLAB ©, a quick way to get started would be to read NumPy for MATLAB users .

In the code samples of this user manual, and in the built-in scripts that are provided with the MLA software, we use the convention that all NumPy commands start with `np.`, as in `y = np.sin(x)` because we `import numpy as np`. Similarly, matplotlib commands can start with `plt.` as in `plt.plot(x,y)` because we `import matplotlib.pyplot as plt`. But with matplotlib we often generate new objects, and work with their member functions, as in the sequence:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,2*np.pi)
y = np.cos(x)
fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
ax1 = fig.add_subplot(2,1,1)
ax1.plot(x,y)
plt.draw()
plt.show()
```

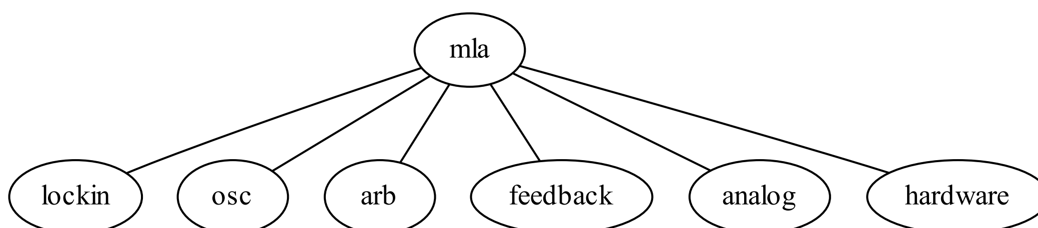
When running integrated MLA scripts matplotlib.pyplot sometimes uses a conflicting GUI back-end, resulting in the rest of the MLA GUI freezing while the figure is displayed. To avoid this problem, plotting can instead be done inside the MLA GUI, from the *Script plot panel* or *Pyhton shell panel*, using the scriptplot object, as demonstrated in the example below.

```
x = np.linspace(0,2*np.pi)
y = np.cos(x)
fig = scriptplot.fig
ax1 = fig.add_subplot(2,1,1)
ax1.plot(x,y)
wx.CallAfter(scriptplot.my_draw)
```

For more details on script plotting inside the GUI, see *plotting in the GUI*.

7.2 MLA API

The Application Programming Interface (API) for the MLA is the main object which we always instantiate with the name mla in Python code. This API object is a container with six basic singleton objects.



Complete documentation of all the member functions of each class is found under the links below.

7.2.1 mla

class mlaapi.mla_api.MLA(*settings*, *calibrations_folder=None*, *eh=None*)

This is the main class for accessing the MLA API. It holds instances of the classes lockin.Lockin, osc. Osc, arb.ARB, feedback.Feedback and hardware.Hardware, which are in turn used for different aspects of setting up and reading data from the MLA. When the MLA class is instanciated, not much happens. Ethernet connection is established in the member function connect() and the MLA is configured and the child objects are created.

Parameters

- settings – a configobj object containing various settings. A settings object can be created by the module impconfig.py. For further information, look at impconfig.py and mla_config_spec.ini.
- calibrations_folder – path to a folder on the local PC where calibrations are stored. If None the temp folder of the operating system will be used. (The calibration is stored in the MLA and coped over to the PC everytime connect() is executed.)

- eh – An eventhandler object that can be used to notify various parts in the software about changes that occur in other parts of the softare. For more information, look at eventhandler.py.

connect(*block=True*, *server_text_callback=None*, *print_callback=None*, *ip_address=None*, *clkref_external=False*, *ping_attempts=100*)

Setup an ethernet connection between the PC and the MLA and configure the MLA.

Parameters

- block – True or False. Weather to block further program execution while the connect function executes. If False, make sure to run mla.connect_thr.join() before any access to the MLA is made.
- server_text_callback – A function taking a string as argument to handle text messages coming from the MLA firmware. If None, python's print() function will be used.
- print_callback – A function taking a string argument and optionally an append_new_line keyword argument to handle messages text messages from the MLA API. If None, python's print() function will be used.
- ip_address – [string or None] The IP address for the MLA to connect to. If None, the first IP address in the list self.settings['COMMUNICATION']['mla_ip'] will be used.
- clkref_external (bool) – if True, set up the clock circuit to use an external 10 MHz reference.

Returns

None

disconnect()

Disconnect the ethernet connection to the MLA. This should be done before the python program exits.

reset(*event=True*, *wait_for_effect=True*)

Convenience function to set all the user-configurable settings to a known initial state.

Parameters

- event (bool, optional) – make the eventhandler trigger the EVT_AMPLITUDES_UPDATED event.
- wait_for_effect (bool, optional) – block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see lockin.start_lockin() and lockin.wait_for_settings_effect().

run_calibration_sequence(*base_calibration=''*)

Start a text based calibration wizard for input and output voltages.

Parameters

base_calibration – Path to calibration file, dictionary with calibration values, or None. The calibration values specified by this paramter will be used when the user skips a calibration step. If '' the calibration factors from calspec.ini will be used.

Returns

None

7.2.2 mla.lockin

class mlaapi.lockin.Lockin(*hardware*)

This class handles communication with and control of the MLA. Most functions set parameters of the measurement and readout data from the MLA.

Many functions has a keyword argument called *unit*. The default is *unit='calibrated'* which will use the current calibration, for more information, see *Calibration*. Instead *unit='digital'* can be selected to communicate with the digital components in their native units. In some functions, other units such as 'percent', 'rad' and 'degree' are also possible.

There are three different methods to wait for new data. While they can be configured to do the same thing, they are different in their implementation:

- wait_for_counter() is the most general function, but it consumers more resources.
- wait_for_new_pixels() is more specific and consumers less resources.
- wait_for_trigger() will consume less resources while waiting, giving improved performance in multi-threaded applications.

abort_parameter_sweep()

Abort a hardware-automated parameter sweep, such as a frequency sweep.

Examples

```
>>> mla.lockin.frequency_sweep(100e3, 200e3, 100)
>>> time.sleep(1)
>>> mla.lockin.abort_parameter_sweep()
```

abort_wait_for_counter()

Aborts the command wait_for_counter() which blocks execution. This abort command must be called from another thread.

Returns
None

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_df(10)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_counter).start()
>>> mla.lockin.wait_for_counter('trig1', increment=1)
-1
>>> mla.lockin.stop_lockin()
```

See also:

wait_for_counter()

abort_wait_for_new_pixels()

Aborts the command wait_for_new_pixels() which blocks execution. This abort command must be called from another thread.

Returns
None

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_new_pixels).start()
>>> mla.lockin.wait_for_new_pixels(100)
>>> mla.lockin.stop_lockin()
```

See also:

wait_for_new_pixels()

abort_wait_for_trigger()

Aborts the command wait_for_trigger() by sending a fake trigger. Since wait_for_trigger() blocks execution, this abort command must be called from another thread. When aborted, the buffer position returned by wait_for_trigger() will be -1.

Returns
None

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_trigger_flanks(trig1_flank='positive', trig2_flank='positive')
>>> mla.lockin.enable_wait_for_trigger(True)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_trigger).start()
>>> mla.lockin.wait_for_trigger()
-1
>>> mla.lockin.stop_lockin()
>>> mla.lockin.flush_trigger_queue()
```

See also:

wait_for_trigger()

apply_settings()

Dummy function for forward compatibility with later API versions

apply_settings_and_wait_for_effect(*timeout=None*)

Dummy function for forward compatibility with later API versions

arm_trigger(*dma_channel=1*, *cluster_size='auto'*)

Arm the lockin trigger so that next time a trigger event occurs on port TRIG IN 1, all waveform and lockin calculations are reset and restared.

Parameters

- dma_channel – 0 or 1
- cluster_size – passed on to start_lockin

Returns
None

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.arm_trigger()
>>> time.sleep(2)
>>> mla.lockin.stop_lockin()
```

Note: Run this command instead of mla.lockin.start_lockin() to start a triggered measurement.

See also:

start_lockin(), stop_lockin().

blank_output(*do_blank*, *wait_for_effect=True*)

Turn on or off all output.

Parameters

- do_blank (boolean) – When True, turns off the output, False turns it on.
- wait_for_effect (boolean) – When True, blocks excecution until a new data packet arrives. Effects only lockin streaming.

Returns
None

See also:

start_lockin() and wait_for_settings_effect().

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.blank_output(True)
>>> mla.lockin.get_pixels(n_pix=1, data_format='IQreal', unit='digital')[0]
array([  2.889704633-05,  -1.908409882-05,  ..., -1.24725527e-05])
>>> mla.lockin.blank_output(False)
>>> mla.lockin.get_pixels(n_pix=1, data_format='IQreal', unit='digital')[0]
array([  2.20439491e+02,  -1.00917580e+02,  ..., -1.47609380e+02])
>>> mla.lockin.stop_lockin()
```

enable_pixel_clock(*enable=True*)

When the pixel clock is enabled a trigger pulse will be sent to TRIG OUT 1 at the start of the each lockin measurement time window.

Parameters
enable (boolean) – enable

Returns
None

Note: Each trigger is 50 clock cycles long, which normally corresponds to 1 microsecond

enable_wait_for_trigger(*state*)

Enables wait_for_trigger() function.

Parameters

- state (boolean) – Ehen True wait_for_trigger() is enabled,

- disabled. (when False) –

Returns
None

> **Warning:** If the data is never read using wait_for_trigger() or flush_trigger_queue() the queue will eventually become over-full and the software will crash.

Details:
This function makes the ethernet packet receiver, pipe the buffer position of every packet where the trigger counter incremented.

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_trigger_flanks(trig1_flank='positive', trig2_flank='positive')
>>> mla.lockin.enable_wait_for_trigger(True)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_trigger).start()
>>> mla.lockin.wait_for_trigger()
187
>>> mla.lockin.stop_lockin()
```

See also:

wait_for_trigger(), flush_trigger_queue()

extract_dc(*meta*, *port=None*, *units='calibrated'*)
Extract DC values from pixel metadata.

Parameters

- meta – meta data object from, for example mla.lockin.get_pixels
- port (integer) – input port number (1–4) or None for all ports
- units (string) – 'calibrated' or 'native'

flush_trigger_queue()
Read out and discard all values in the trigger queue (i.e. buffer positions of packets with incremented trigger count).

Returns
None

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_trigger_flanks(trig1_flank='positive', trig2_flank='positive')
>>> mla.lockin.enable_wait_for_trigger(True)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_trigger).start()
>>> mla.lockin.wait_for_trigger()
-1
>>> mla.lockin.stop_lockin()
>>> mla.lockin.flush_trigger_queue()
```

See also:

enable_wait_for_trigger(), wait_for_trigger()

frequency_sweep(*f1*, *f2*, *df_in=1000*, *pts=1001*, *pixels_per_step=5*)

Start a frequecy sweep using one single lockin.

Parameters

- f1 (float) – [Hertz], start frequency of the sweep.
- f2 (float) – [Hertz], stop frequency of the sweep.
- df_in (float) – [Hertz], requested measurement bandwidth.
- pts (integer) – number of steps in the sweep.
- pixels_per_step (integer) – integer, number of measurements performed at each step.

Returns

None

Details:

This function uses hardware-enabled functionality for a performing a fast parameter sweep without having to realy on communication between the computer and the MLA between each step in the sweep. This function only starts the sweep. Use get_frequency_sweep() to read out the results. The sweep is not continuous but in discret steps, taken so as to avoid Fourier leakage (i.e. the lock-in measurement on an an integer number of periods). Due to this tuning at each step, the measurement bandwidth and the frequency steps are not exactly uniform, but sufficiently close for most applications. In high-Q systems it takes longer time to achieve stedady-state oscillation after a step. Use pixels_per_step to slow down the sweep and avoid transients. Only the last pixel will be used at each frequency step. The settling time before measurement is then (pixels_per_step - 1)/df.

See also:

get_frequency_sweep() frequency_sweep_multifreq() abort_parameter_sweep()

frequency_sweep_multifreq(*f1*, *f2*, *df_in=1000*, *pts=1001*, *pixels_per_step=5*)

Start a frequecy sweep using all available lockins simultaneousely.

Parameters

- f1 (float) – [Hertz], start frequency of the sweep.
- f2 (float) – [Hertz], stop frequency of the sweep.
- df_in (float) – [Hertz], requested measurement bandwidth.
- pts (integer) – number of steps in the sweep.
- pixels_per_step (integer) – number of measurements performed at each step.

Returns

None

Details:

Similar to frequency_sweep(), but uses all available lockins in parallel. With multifrequency a sweep can be performed much faster but it will only give the same result as a single-freqeuncy sweep if the system is linear. A nonlinear system will intermodulate the tones of a multifrequency drive.

See also:

get_frequency_sweep_multifreq() frequency_sweep() abort_parameter_sweep()

get_Tm()

Get measurement time window, inverse of measurement bandwidth i.e. Tm=1/df

Returns
measurement time window

Return type
float [Seconds]

Examples

```
>>> mla.lockin.set_df(500)
>>> mla.lockin.get_Tm()
0.002
```

See also:

get_samples_per_pixel() set_samples_per_pixel() set_Tm() set_df()

get_Tm_max()

Get longest allowed measurement time window

Returns
longest possible measurement time window.

Return type
float [Seconds]

See also:

get_Tm_min()

get_Tm_min()

Get shortest allowed measurement time window

Returns
shortest possible measurement time window.

Return type
float [Seconds]

See also:

get_Tm_max()

get_amplitudes(*unit='calibrated'*)

Return the output amplitudes.

Parameters
unit – {'calibrated', 'digital', 'percent' , '%'} unit of given amplitudes.

Returns
output amplitudes

Return type
np.array(dtype=float)

See also:

set_amplitudes()

get_counter(*counter='trig1'*, *bufpos=None*)

Return the value of the specified counter at the specified position in the lockin data buffer.

Parameters

- counter (integer or string) – must be one of the following:

1 = 'settings': The settings counter increments every time a parameter is changed. This counter can be used to identify which data had which parameters. Note that the frequency and phase parameters will increment the settings counter by 2 when they are changed.

2 = 'trig2' = 'image_nr'
[The trig2 counter counts the number of trigger] events recieved at port TRIG IN 2 (image trigger for SPM).

3 = 'trig1' = 'line_nr': The trig1 counter counts the number of trigger events recived at in TRIG IN 1 (line trigger for SPM).

4 = 'pixel': The pixel counter counts the total number of time measurement windows
that has passed since the MLA was powered on. NOTE: this counter will continue to increase regardless of weather the data was transmitted to the computer or not.

- bufpos – integer or None, the buffer position where to read the value. If bufpos is left at None the last value will be used.

Returns
The value of the counter

Return type
integer

Examples

```
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.start_lockin()
>>> mla.lockin.get_counter('pixel')
24
>>> mla.lockin.wait_for_counter('pixel', increment=5)
29
>>> mla.lockin.get_counter('pixel')
29
```

get_da_max()

Get largest possible amplitude in native lockin units

Returns
The highest possible output amplitude in native lockin units.

Return type
float

Note: Even though the DA converter has 16 bits the resolution of lockin drive amplitudes is significantly higher. This is necessary to allow one drive tone to fill upp the entire DA range, and, at the same time, allow multiple tones of very low amplitude to still yeald a low voltage even when interfering constructively.

See also:

get_da_min()

get_da_min()

Get lowest possible amplitude in native lockin units

Returns
The lowest (most negative) possible output amplitude in native lockin units.

Return type
float

Note: Even though the DA converter has 16 bits the resolution of lockin drive amplitudes is significantly higher. This is necessary to allow one drive tone to fill upp the entire DA range, and, at the same time, allow multiple tones of very low amplitude to yeild a low voltage even when interfering constructively.

See also:

get_da_max()

get_df()

Get measurement bandwidth

Returns
measurement bandwidth in Hertz, or inverse of measurement time window, df=1/Tm.

Return type
float [Hertz]

Examples

```
>>> mla.lockin.set_Tm(0.001)
>>> mla.lockin.get_df()
1000.0
```

See also:

get_samples_per_pixel() set_samples_per_pixel() set_Tm() set_df()

get_df_max()

Get largest allowed measurement bandwidth

Returns
The largest possible measurement bandwidth.

Return type
float [Hertz]

See also:

get_df_min()

get_df_min()

Get smallest allowed measurement bandwidth

Returns
The smallest possible measurement bandwidth.

Return type
float [Hertz]

See also:

get_df_max()

get_frequencies()

Returns
frequencies of all tones in Hertz.

Return type
np.array(dtype=float) [Hertz]

Examples

```
>>> mla.lockin.set_frequencies(100e3)
>>> mla.lockin.get_frequencies()
array([ 100000.00002037,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ,
              0.        ,        0.        ,        0.        ])
```

```
>>> mla.lockin.set_frequencies(100e3+np.arange(mla.lockin.nr_input_freq)*1e3)
>>> mla.lockin.get_frequencies()
array([ 100000.00002037,  100999.99999511,  102000.00001532,
        102999.99999006,  104000.00001027,  104999.99998501,
        106000.00000522,  106999.99997996,  108000.00000017,
        109000.00002039,  109999.99999513,  111000.00001534,
        111999.99999008,  113000.00001029,  113999.99998503,
        115000.00000524,  115999.99997998,  117000.00000019,
        118000.0000204 ,  118999.99999514,  120000.00001535,
        120999.99999009,  122000.0000103 ,  122999.99998504,
        124000.00000525,  124999.99997999,  126000.0000002 ,
        127000.00002042,  127999.99999515,  129000.00001537,
        129999.9999901 ,  131000.00001032,  131999.99998506,
        133000.00000527,  133999.99998001,  135000.00000022,
        136000.00002043,  136999.99999517,  138000.00001538,
        138999.99999012,  140000.00001033,  140999.99998507])
>>> mla.lockin.set_frequencies(300e3+np.arange(10)*1e3, idx=range(10))
>>> mla.lockin.get_frequencies()
array([ 300000.00001564,  300999.99999038,  302000.00001059,
        302999.99998533,  304000.00000554,  304999.99998028,
        306000.00000049,  307000.00002071,  307999.99999545,
        309000.00001566,  109999.99999513,  111000.00001534,
        111999.99999008,  113000.00001029,  113999.99998503,
        115000.00000524,  115999.99997998,  117000.00000019,
        118000.0000204 ,  118999.99999514,  120000.00001535,
        120999.99999009,  122000.0000103 ,  122999.99998504,
        124000.00000525,  124999.99997999,  126000.0000002 ,
        127000.00002042,  127999.99999515,  129000.00001537,
        129999.9999901 ,  131000.00001032,  131999.99998506,
        133000.00000527,  133999.99998001,  135000.00000022,
        136000.00002043,  136999.99999517,  138000.00001538,
        138999.99999012,  140000.00001033,  140999.99998507])
```

See also:

set_frequencies()

get_frequency_sweep(*unit='calibrated'*)

Extract the results of a frequency sweep started by frequency_sweep(). If executed before sweep is

finished, will return partial sweep.

Parameters
unit (string) – 'calibrated' or 'digital'

Returns
(f, a, n)

- f : np.array(dtype=float) [Hertz] frequencies of sweep.
- a : np.array(dtype=float) [ADU] quadrature amplitudes.
- n : integer, length of the f and a arrays.

Return type
Tupel

See also:

frequency_sweep(), *Calibration*

get_frequency_sweep_multifreq(*unit='calibrated'*)

Extract the reults of a frequency sweep started by frequency_sweep_multifreq(). If executed before sweep is finished, will return partial sweep.

Parameters
unit (string) – ('calibrated', 'digital')

Returns
(f, a, n)

- f : np.array(dtype=float) [Hertz] frequencies of sweep.
- a : np.array(dtype=complex) [unit] quadrature amplitudes.
- n : integer, length of the f and a arrays.

Return type
Tuple

See also:

frequency_sweep_multifreq(), *Calibration*

get_fsample()

Get sampling frequency

Returns
The sampling frequency in Hertz.

Return type
float [Hertz]

Note: The lockin unit is clocked by the AD-converter clock. In almost all cases the DA clock should be set to have the same frequency.

See also:

hardware.Hardware.set_adc_clock_divisor()

get_input_multiplexer()

Returns which input port is connected to which demodulator

Returns
list of integers with port numbers 1-4.

Return type
port_array

See also:

set_input_multiplexer()

get_line(*line_length_if_first_trig=1*, *data_format='complex'*, *unit='calibrated'*, *get_meta=False*)

Return all the lockin packets between two successive trigger events on port TRIG IN 1.

Parameters

- line_length_if_first_trig – If there is only one trigger event in the lockin data buffer, this parameter specifies how many pixels to return.
- data_format – {'IQreal', 'complex', 'amp', 'phase'} specifies return data type.
- unit – {'calibrated', 'digital', 'deg, 'rad'} specifies units. 'rad' and 'deg' only for phase.

Returns

Numpy array of pixel data as specified by the data_format and unit parameters.

Note: Other trigger methods are possile with firmware modificaitons. Please contact Intermodulation Products if you need different trigger methods.

Examples

```
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.start_lockin()
>>> mla.lockin.wait_for_trigger()
>>> mla.lockin.wait_for_trigger()
>>> mla.lockin.get_line()
```

See also:

wait_for_trigger(), *Calibration*

get_new_pixels(*throw_away=False*, *wait_for_new=True*, *rawdata=False*)

Return all new lockin packets from the buffer.

Parameters

- throw_away – boolean (default=False) when True, all previous data will be discarded
- wait_for_new – boolean (default=True) when True, will block program excutation until at least one new data packet arrives.
- rawdata – Boolean (default=False) when True, returns raw Fourier sums, not devied by the number of samples and not phase corrected.

Returns

(pixels, nr_pixels, meta)

- **pixels**
  [two dim. np.array(dtype=float) [ADU]] axis 0: [I_0, Q_0, I_1, Q_1, I_2, Q_2, . . .] axis 1: [pixel_0, pixel_1, pixel_2, . . .]
- nr_pixels : integer, number of pixels returned (i.e. length of axis 0)
- **meta**
  [two dim. np.array(dtype=uint32)] axis 0: [header, settings_counter, trig2 counter, trig1 counter, pixel counter, trigger position, dc_data] axis 1: [pixel_0, pixel_1, pixel_2, . . .]

Return type
Tuple

Usage:
Keeps track of which data has been extracted from the buffer. This function only returns NEW data that was not previously extraced. Usually issued in a loop together with plotting and other data processing commands.

get_parameter_sweep(*unit='calibrated'*)

Extract the reults of a parameter sweep started by parameter_sweep(). If executed before sweep is finished, will return partial sweep.

Parameters
unit (string) – ('calibrated', 'digital')

Returns
(pixels, meta)

Return type
Tuple

See also:

frequency_sweep_multifreq(), *Calibration*

get_phases(*unit='rad'*)

Return the output phases. :param unit: 'rad' or 'degree'

Returns
output phases

Return type
np.array(dtype=float)

Examples

```
>>> mla.lockin.set_phases([np.pi/6]*np.arange(13))
>>> mla.lockin.get_phases('degree')[:13]
array([ 0., 30., 60., 90., 120., 150., 180., 210., 240., 270., 300., 330., 360.])
>>> mla.lockin.set_phases(30*np.arange(13), 'degree')
>>> mla.lockin.get_phases('rad')[:13]
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
5.23598776,  5.75958653,  6.28318531])
```

See also:

set_phases()

get_pixel_average(*n_pix*, *data_format='complex'*, *unit='calibrated'*, *first_bufpos=None*)

Extract a pixel as an average of n measured pixels

Parameters

- n_pix (integer) – number of measurement points to extract. Set to None to extract all pixels from the first_bufpos to the end.
- data_format ('IQreal', 'complex', 'amp', 'phase') – specifies return data type.
- unit ('calibrated', 'digital', 'deg, 'rad') – specifies returned units. 'calibrated' is typically volts. 'rad' and 'deg' only for phase.

- first_bufpos (integer) – Specify the first buffer position to extract data from. This can be useful when extracting data with respect to a trigger position. Leave at None to extract the n_pix last pixels.

Returns

(pixels, meta)

- pixels : np.array(dtype=np.float64) of pixels of the specified data_format and unit.
- **meta**
  [two dim. np.array(dtype=np.uint32)] axis 0: [header, settings_counter, trig2 counter, trig1 counter, pixel counter, trigger position, dc_data] axis 1: [pixel 1, pixel 2, pixel 3, …]

Return type
Tuple

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.start_lockin()
>>> mla.lockin.wait_for_new_pixels(10)
>>> pixels, meta = mla.lockin.get_pixel_average(10)
>>> pixels.shape
(42L, 1L)
>>> meta.shape
(12L, 10L)
>>> mla.lockin.stop_lockin()
```

See also:

mlaapi.lockin.LockinReceiver.get_new_pixels(), get_tone(), get_line(), get_pixel_average(), get_pixels(), *Calibration*

get_pixels(*n_pix=None*, *data_format='complex'*, *unit='calibrated'*, *first_bufpos=None*)

Extract pixels from the lockin data buffer.

Parameters

- n_pix (integer) – number of pixels to extract. Set to None to extract all pixels from the first_bufpos to the end.
- data_format ('IQreal', 'complex', 'amp', 'phase') – specifies return data type.
- unit ('calibrated', 'digital', 'deg, 'rad') – specifies returned units. 'calibrated' is typically volts. 'rad' and 'deg' only for phase.
- first_bufpos (integer or None) – The first buffer position to extract data from. This can be useful when extracting data with respect to a trigger position. Leave at None to extract the n_pix last pixels.

Returns

(pixels, meta)

- **pixels**
  [np.array(dtype=float) 2D array of pixel data in specified data_format and unit.] axis 0: [tone_0 data, tone_1 data, tone_2 data…] axis 1: [pixel_0, pixel_1, pixel_2, …]
- **meta**
  [np.array(dtype=uint32)] axis 0: [header, settings_counter, trig2 counter, trig1 counter, pixel counter, trigger position, dc_data] axis 1: [pixel_0, pixel_1, pixel_2, …]

Return type
Tuple

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.start_lockin()
>>> mla.lockin.wait_for_new_pixels(10)
>>> pixels, meta = mla.lockin.get_pixels(10)
>>> pixels.shape
(42L, 10L)
>>> meta.shape
(12L, 10L)
>>> mla.lockin.stop_lockin()
```

See also:

mlaapi.lockin.LockinReceiver.get_new_pixels(), get_tone(), get_line(), get_pixel_average(), *Calibration*

get_pixels_between_bufpos(*first_bufpos*, *last_bufpos*, *data_format='complex'*, *unit='calibrated'*)

Extract pixels from the lockin data buffer between two given buffer positions.

Parameters

- first_bufpos (integer) – a position in the pixel buffer corresponding to the ealiest data to extract
- last_bufpos (integer) – a position in the pixel buffer corresponding to a later position. Data upto, but not including, this position is extracted.
- data_format ('IQreal', 'complex', 'amp', 'phase') – specifies return data type.
- unit ('calibrated', 'digital', 'deg, 'rad') – specifies returned units. 'calibrated' is typically volts. 'rad' and 'deg' only for phase.

Returns

(pixels, meta)

- **pixels**
  [np.array(dtype=float) 2D array of pixel data in specified data_format and unit.] axis 0: [tone_0 data, tone_1 data, tone_2 data...] axis 1: [pixel_0, pixel_1, pixel_2, ...]
- **meta**
  [np.array(dtype=uint32)] axis 0: [header, settings_counter, trig2 counter, trig1 counter, pixel counter, trigger position, dc_data] axis 1: [pixel_0, pixel_1, pixel_2, ...]

Return type
Tuple

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.start_lockin()
>>> first_bufpos = 0
>>> last_bufpos = 10
>>> pixels, meta = mla.lockin.get_pixels(first_bufpos, last_bufpos)
>>> pixels.shape
(42L, 10L)
>>> meta.shape
(12L, 10L)
>>> mla.lockin.stop_lockin()
```

See also:

get_line(), get_pixels(), wait_for_trigger()

get_port(*tone_index*)

Parameters
tone_index (integer) – index of the measured tone.

Returns
port number where tone is measured.

Return type
integer

get_samples_per_pixel()

Returns
ns = the number of samples in measurement time window.

Return type
integer

The number of samples per pixel (ns) is the fundamental hardware unit for the measurement time window (Tm), or measurement bandwidth (df=1/Tm). With sampling frequency fs, ns = fs*Tm = fs/df

Examples

```
>>> mla.lockin.set_Tm(0.001)
>>> mla.lockin.get_samples_per_pixel()
50000
```

See also:

get_df() set_samples_per_pixel() set_Tm() set_df()

get_samples_per_pixel_max()

Get largest allowed samples per pixel

Returns
largest allowed samples_per_pixel.

Return type
int

See also:

get_samples_per_pixel_min() get_df_max() get_df_min() get_Tm_max() get_Tm_min()

get_samples_per_pixel_min()

Get smallest allowed samples per pixel

Returns
smallest allowed samples_per_pixel.

Return type
int

See also:

get_samples_per_pixel_max() get_df_max() get_df_min() get_Tm_max() get_Tm_min()

get_tone(*tone_index=0*, *n_points=None*, *data_format='complex'*, *unit='calibrated'*, *first_bufpos=None*)

Extract n_points of lockin measurement data of a single tone. The last item in the returned list will be the last data point that has arrived to the computer.

Parameters

- n_points – integer number of measurement points to extract
- tone_index – integer, index of the tone to extract (use tone_index=0 for the first tone)
- data_format – {'IQreal', 'complex', 'I', 'Q', 'amp', 'phase'} specifies return data type.
- unit – {'calibrated', 'digital', 'deg, 'rad'} specifies units. 'rad' and 'deg' only for phase.

Returns
Array of n lockin measurement data points of the specified unit and data_format.

Return type
np.array

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.set_df(100)
>>> mla.lockin.set_amplitudes([2]*mla.lockin.nr_output_freq, unit='%')
>>> mla.lockin.set_phases(0)
>>> mla.lockin.set_frequencies(100e3 + np.arange(mla.lockin.nr_input_freq)*1e3)
>>> mla.lockin.get_tone(n_points=3,  tone_index=0, data_format='complex', unit=
↪'calibrated')
array([ 0.00868193-0.00263251j,  0.00868193-0.00263251j,
        0.00868193-0.00263251j])
>>> mla.lockin.get_tone(n_points=3,  tone_index=0, data_format='amp', unit=
↪'calibrated')
array([ 0.00907227,  0.00907227,  0.00907227])
>>> mla.lockin.get_tone(n_points=3,  tone_index=0, data_format='phase', unit='deg')
array([-16.86823347, -16.86823347, -16.86823347])
>>> mla.lockin.get_tone(n_points=3,  tone_index=0, data_format='phase', unit='rad')
array([-0.29440621, -0.29440621, -0.29440621])
>>> mla.lockin.get_tone(n_points=3,  tone_index=0, data_format='IQreal', unit=
↪'digital')
array([ 284.48073199,  -86.25959969,  284.48073199,  -86.25959969,
        284.48073199,  -86.25959969])
>>> mla.lockin.stop_lockin()
```

See also:

get_pixels() mlaapi.lockin.LockinReceiver.get_new_pixels(), get_line(), get_pixel_average(), get_pixels(), *Calibration*

is_started()

Check if MLA is sending lockin data to the computer.

Returns
boolean

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.is_started()
True
>>> mla.lockin.stop_lockin()
>>> mla.lockin.is_started()
False
```

line_up_phases(*tones*, *value*)

Calibrate phase measurements at specified **tones** to a particular **value**.

Parameters

- tones (np.array(dtype=int)) – indices of tones to be lined up.
- value (float) – [radians] phase that MLA measures after line-up.

Returns
None

Usage:
The lockin measurement of phase is affected by several frequency dependent things, for example digital delays, analog anti-alias filters, cables, the experimental setup etc. This function should be run after the frequencies and amplitudes have been set up. The input and output ports must also be connected, typically through the experiment. After this function has been executed, the phase is set to **value**. Future phase measurements reflect deviation from **value**.

Examples

```
>>> mla.lockin.set_frequencies(np.arange(mla.lockin.nr_input_freq)*100e3)
>>> mla.lockin.set_amplitudes([0.01]*mla.lockin.nr_input_freq)
```

load_from_file(*path*)

Load the lockin setup from file.

Parameters
path (string) – full path to the file from which the lockin setup should be loaded.

Returns
None

See also:

save_to_file()

logger_start(*filename*, *starttime=None*, *flush_interval=1*)

Start logging lockin data to file.

Parameters

- filename – path to the log file
- starttime – None or datetime (as returned by eg. datetime.datetime.now()). Time to write out as the Start time: in the log file header. If None, the local time of the computer will be used.
- flush_interval – float [seconds] How often should the log file be written to disk.

Returns
None

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.locker_start('testfila.txt')
>>> mla.lockin.start_lockin()
>>> time.sleep(2)
>>> mla.lockin.logger_stop()
>>> mla.lockin.stop_lockin()
```

See also:

logger_stop()

logger_stop(*stop_time=None*)

Stop logging lockin data to file.

Parameters
stop_time – None or datetime (as returned by eg. datetime.datetime.now()). Time to write out as the Stop time: at the end of the file. If None, the local time of the computer will be used.

Examples

```
>>> mla.lockin.set_df(10)
>>> mla.lockin.locker_start('testfila.txt')
>>> mla.lockin.start_lockin()
>>> time.sleep(2)
>>> mla.lockin.logger_stop()
>>> mla.lockin.stop_lockin()
```

See also:

logger_stop()

measure_packet_rate(*measurement_time=1*)

Measure the rate at which lockin packets are arriving to the computer.

Parameters
measurement_time – float (seconds)

Returns
float [number of packets per second]

Note: This functions is used mostly for debugging and testing.

---

reset_outputs(*event=True*, *wait_for_effect=True*)

Set all output masks, amplitudes and dc offsets to zero.

Parameters

- event (boolean) – Make the eventhandler trigger the EVT_AMPLITUDES_UPDATED event.
- wait_for_effect (boolean) – Block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see start_lockin() and wait_for_settings_effect().

reset_tones_on_new_pixel(*reset=True*)

Turn on or off the phase reset at the start of each measurement time window.

Parameters
reset (boolean) – False turns off the phase reset.

Returns
None

Usage:
Corrects for slow phase drift of lockin measuremnet of intermodulaiton products when tuning is not perfect. The reset is typically turned off only when you want to generate a tone with a very exact frequency (very stable phase) and you are not interested in measuring intermodulation products.

See also:

reset_tones_on_trig1()

reset_tones_on_trig1(*reset=True*)

Turn on or off the phase reset when a tigger event on port TRIG IN 1.

Parameters
reset (boolean, defalut=True) – Flase turns off the phase reset

Returns
None

See also:

reset_tones_on_new_pixel()

save_to_file(*path*)

Save the lockin setup to a file.

Parameters
path (string) – The full path of the file to be saved

Returns
None

See also:

load_from_file()

send_trigger()

Sends a single trigger event to the port TRIG OUT 1

Examples

```
>>> mla.lockin.send_trigger()
```

Note: The TRIG OUT 1 will go high for 100-1000 us and then become low.

set_Tm(*Tm*, *wait_for_effect=True*)

Parameters

- Tm – float [seconds] measurement time window
- wait_for_effect – bolean (default=True) when true, blocks execution until recieved lockin data reflects the action of this command.

Returns
None

Examples

```
>>> mla.lockin.set_Tm(0.001)
>>> mla.lockin.get_Tm()
0.001
```

See also:

get_samples_per_pixel() set_samples_per_pixel() set_Tm() set_df()

set_amplitudes(*amplitudes*, *unit='calibrated'*, *idx='all'*, *port=None*, *event=True*, *wait_for_effect=True*)

Sets the amplidues of the output tones. If the amplitudes are complex, the np.angle() function is used to set the phase. If amplitudes are pure real, the phases will be unchanged.

Parameters

- amplitudes – float, array_like, real or complex.
- unit – {'calibrated', 'digital', 'percent' , '%'} unit of given amplitudes.
- idx – array_like or 'all'. Specifices indicies of tones for which amplitudes should be set. If 'all' is given, and the number of tones is larger than the length of the list, the amplitudes of the remaining tones are set to 0.
- port – array_like or None. Specifies a port for calibration V to ADU. If None, the port calibration is determined from the output_mask. If both ports are specified in the output_mask, the first port is used. This parameter does not change the output_mask.
- event – True or False (default=True). Make the eventhandler trigger the EVT_AMPLITUDES_UPDATED event.
- wait_for_effect – True or False (default=True). Block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see start_lockin() and wait_for_settings_effect().

Returns
None

Examples

```
>>> mla.lockin.set_amplitudes(1) # Sets the first amplitude to 1V and the rest to 0%
>>> mla.lockin.set_amplitudes([2.5]*10, unit='%', idx=range(10)) # Set the first 10␣
↪amplitudes to 2.5%, leave the rest unchanged
```

See also:

wait_for_settings_effect(), set_output_mask(), *Calibration*

set_dc_offset(*port*, *value*, *unit='calibrated'*)

Apply a DC offset to the generated output waveforms.

Parameters

- port – 1 or 2, specifies output port to which dc offset is applied.
- value – float, value to ouput.
- unit – 'calibrated' or 'digital' (-32768 to 32767).

Returns
None

> **Warning:** No check is made weather the sum of all waveforms and the DC offset lies within the valid range. The waveform will wrap from max (min) to zero if value is out of range.

Examples

```
>>> mla.lockin.set_dc_offset(1, 0.7)
```

See also:

set_amplitudes(), *Calibration*

set_df(*df*, *event=True*, *wait_for_effect=True*)

Parameters

- df – [Hz] measurement bandwidth, inverse of measurement time window df=1/Tm
- wait_for_effect – bolean (default=True) when true, blocks execution until recieved lockin data reflects the action of this command.

Returns
None

Examples

```
>>> mla.lockin.set_df(1000.)
>>> mla.lockin.get_df()
1000.0
```

See also:

get_samples_per_pixel() set_samples_per_pixel() set_Tm() set_df()

set_digital_mask(*mask*)

Set a mask for outputting the last three frequencies as square waves on ports TRIG OUT 1, TRIG OUT 2 and TRIG OUT 3.

Parameters
mask – list or integer mask

Returns
None

Examples

Turn on the last frequency to TRIG OUT 3 >>> mla.lockin.set_digital_mask([0, 0, 1])

Turn on the second last frequency to TRIG OUT 2 >>> mla.lockin.set_digital_mask([0, 1, 0])

Turn on the third last frequency to TRIG OUT 1 >>> mla.lockin.set_digital_mask([1, 0, 0])

Turn on all three at the same time >>> mla.lockin.set_digital_mask([1, 1, 1])

Turn off all digital waveforms >>> mla.lockin.set_digital_mask(0)

set_frequencies(*f*, *unit='Hz'*, *idx='all'*, *event=True*, *wait_for_effect=True*)

Parameters

- f – np.array(dtype=float) [Hertz] frequencies of all tones.
- idx – array_like or 'all', indicies of chanels to set frequencies If 'all' is given and number of frequencies is higher than the length of idx, remaining amplitudes are set to 0.
- event – boolean (default=True) when True the eventhandler triggers EVT_AMPLITUDES_UPDATED event.
- wait_for_effect – bolean (default=True) when true, blocks execution until command has taken effect.

Returns
None

Examples

```
>>> mla.lockin.set_frequencies(100e3)
>>> mla.lockin.get_frequencies()
array([ 100000.00002037,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ,
             0.        ,      0.        ,      0.        ])
```

```
>>> mla.lockin.set_frequencies(100e3+np.arange(mla.lockin.nr_input_freq)*1e3)
>>> mla.lockin.get_frequencies()
array([ 100000.00002037,  100999.99999511,  102000.00001532,
        102999.99999006,  104000.00001027,  104999.99998501,
        106000.00000522,  106999.99997996,  108000.00000017,
```

(continues on next page)

(continued from previous page)

```
       109000.00002039,  109999.99999513,  111000.00001534,
       111999.99999008,  113000.00001029,  113999.99998503,
       115000.00000524,  115999.99997998,  117000.00000019,
       118000.0000204 ,  118999.99999514,  120000.00001535,
       120999.99999009,  122000.0000103 ,  122999.99998504,
       124000.00000525,  124999.99997999,  126000.0000002 ,
       127000.00002042,  127999.99999515,  129000.00001537,
       129999.9999901 ,  131000.00001032,  131999.99998506,
       133000.00000527,  133999.99998001,  135000.00000022,
       136000.00002043,  136999.99999517,  138000.00001538,
       138999.99999012,  140000.00001033,  140999.99998507])
>>> mla.lockin.set_frequencies(300e3+np.arange(10)*1e3, idx=range(10))
>>> mla.lockin.get_frequencies()
array([ 300000.00001564,  300999.99999038,  302000.00001059,
       302999.99998533,  304000.00000554,  304999.99998028,
       306000.00000049,  307000.00002071,  307999.99999545,
       309000.00001566,  109999.99999513,  111000.00001534,
       111999.99999008,  113000.00001029,  113999.99998503,
       115000.00000524,  115999.99997998,  117000.00000019,
       118000.0000204 ,  118999.99999514,  120000.00001535,
       120999.99999009,  122000.0000103 ,  122999.99998504,
       124000.00000525,  124999.99997999,  126000.0000002 ,
       127000.00002042,  127999.99999515,  129000.00001537,
       129999.9999901 ,  131000.00001032,  131999.99998506,
       133000.00000527,  133999.99998001,  135000.00000022,
       136000.00002043,  136999.99999517,  138000.00001538,
       138999.99999012,  140000.00001033,  140999.99998507])
```

See also:

get_frequencies() wait_for_settings_effect()

set_frequencies_by_n_and_df(*n*, *df*, *tune=False*, *wait_for_effect=True*)

An alternative way to setup the frequencies of the tones with index i, such that f_i=n_i*df, where the measurement bandwidth df and the number of oscillations of each tone n_i, are given instead of the actual frequencies. This method ensures that that the tones are tuned, which is required to avoid Fourier leakage.

Parameters

- n – array of integers sepcifying the number of oscillations of each tone during the measurement time window Tm=1/df.
- df – float [Hertz] desired measurement bandwidth.
- tune – Boolean (default=False) True enforces perfect tuning, (i.e. forces the number of samples per pixel is integer power of two).
- wait_for_effect – Boolean (default=True). True will halt execution until a data packet has arrived where the new setting has taken effect. This option only has effect if the lockin is streaming, see start_lockin() and wait_for_settings_effect().

Returns

(samples_per_pixel, df_tuned, f_hz)

- samples_per_pixel : integer, number of samples in the measurement time window, ns=fs*Tm.
- df_tuned : float [Hertz], the tuned measurement bandwidth.
- f_hz : list of floats [Hertz], the tuned frequencies.

Return type
Tuple

set_input_multiplexer(*port_array*, *event=True*)

Program the multiplexer that controls which input port should be connected to which demodulator.

Parameters

- port_array – list of integers with port numbers 1-4. The length of the list should equal the number of input frequencies.
- event – True or False. Make the eventhandler trigger the EVT_MASK_UPDATED event.

Returns
None

Examples

```
>>> # Only use port 1
>>> mla.lockin.set_input_multiplexer([1]*mla.lockin.nr_input_freq)
>>> # Use ports 1 and 2 alternating
>>> mla.lockin.set_input_multiplexer([1, 2]*mla.lockin.nr_input_freq/2)
```

See also:

get_input_multiplexer() set_output_mode_lockin() feedback.Feedback.set_input_multiplexer()

set_output_mask(*mask*, *port=1*, *event=True*, *wait_for_effect=True*)

Each of the two lockin output ports has a separate mask. Each bit in the mask determines weather the corresponing tone should be added to the output. The same tone can be output on port 1, port 2, or both.

Parameters

- mask – list of 0 or 1, Boolean array, or single integer mask.
- port – 1 or 2
- event – True or False. Make the eventhandler trigger the EVT_MASK_UPDATED event.
- wait_for_effect – True or False. Block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see start_lockin()

Returns
None

Note: All frequencies will always have the same number of samples per pixel.

Examples

```
>>> mla.lockin.set_output_mask([1]*mla.lockin.nr_output_freq, port=1) # Output all␣
↪tones on port 1
```

```
>>> #Generate three tones with 100 kHz, 200 kHz and 300 kHz. Output to port 1 with
>>> # amplitude 0.1 V and to port 2 with amplitude 0.5 V.
>>> mla.lockin.set_output_mask([0, 1, 0, 1, 0, 1], port=1)
>>> mla.lockin.set_output_mask([1, 0, 1, 0, 1, 0], port=2)
>>> mla.lockin.set_frequencies([100e3, 100e3, 200e3, 200e3, 300e3, 300e3])
>>> mla.lockin.set_amplitudes([0.1, 0.5, 0.1, 0.5, 0.1, 0.5])
```

```
>>> mla.lockin.set_output_mask(0xF, port=1) # output tone 0,1,2,3 to port 1
```

set_output_mode_lockin(*two_channels=False*)

Set the fpga output multiplexer to output data from the lockin waveform generators, as opposed to the arbitrary waveform generators.

Parameters

two_channels – [Boolean] True will use both output ports for lockin waveform output. False will use the second port for feedback

set_phases(*p*, *unit='rad'*, *idx='all'*, *event=True*, *correct_phases=True*, *wait_for_effect=True*)

Set the output phases.

Parameters

- p – float, array_like, the phases to be set
- unit – 'rad' or 'degree'
- idx – array_like or 'all'. Specifices which amplitudes should be set. If 'all' is given, and the number of frequencies are higher than the length of the list, the remaining amplitudes are set to 0.
- event – True or False (default=True). Make the eventhandler trigger the EVT_AMPLITUDES_UPDATED event.
- wait_for_effect – True or False (default=True). Block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see start_lockin() and wait_for_settings_effect().

Returns

None

Note: The same waveform is used to both generate and demodulate the signal. Therefore, changing the phase of a tone does not change the native measurement of the phase. However, this effect is compensated for in the PC backend.

Examples

```
>>> mla.lockin.set_phases(np.pi/4) # Sets the first phase to pi/4 radians and the rest to␣
↪0
>>> mla.lockin.set_phases(45, unit='degree') # Sets the first phase to 45 degrees and␣
↪the rest to 0
>>> mla.lockin.set_phases([30]*10, unit='degree', idx=range(10)) # Set the first 10␣
↪phases to 30 degrees, leave the rest unchanged.
```

See also:

get_phases(). wait_for_settings_effect()

set_samples_per_pixel(*n*, *event=True*, *wait_for_effect=True*)

Set the number of samples used to integrate one lockin reading. The integration time will then be T = samples_per_pixel*sampling_frequency

Parameters

- n – integer. the number of samples
- event – True or False. Make the eventhandler trigger the EVT_SAMPLES_PER_PIXEL_UPDATED event.
- wait_for_effect – True or False. Block the execution until a data packet has arrived where the new setting has had effect. This option only has effect of the lockin is streaming, see start_lockin()

Returns
None

Note: All frequencies will always have the same number of samples per pixel.

Examples

```
>>> mla.lockin.set_samples_per_pixel(50000)
>>> mla.lockin.get_df()
1000.0
```

set_trigger_flanks(*trig1_flank='negative'*, *trig2_flank='negative'*)

Warning: This function is deprecated. Use set_trigger_in_flank instead.

Sets trigger flank for hardware to react. For AFM, trig1 and trig2 are end-of-line (EOL) and end-of-frame (EOF) respectively.

Parameters

- trig1_flank – 'positive', 'negative' or 'both'. Defalut is negative.
- trig2_flank – 'positive', 'negative' or 'both'. Defalut is negative.

Returns
None

Examples

```
>>> mla.lockin.set_trigger_flanks(trig1_flank='positive', trig2_flank='positive')
>>> mla.lockin.set_trigger_flanks(trig1_flank='negtive', trig2_flank='both')
```

See also:

wait_for_trigger()

set_trigger_in_flank(*trig_in_port*, *flank*)

Sets which type of flank that should be considered a trigger.

Parameters

- trig_in_port – 1 or 2.
- flank – 'none', 'negative', 'positive' or 'both'

Examples

```
>>> mla.lockin.set_trigger_flanks(1, 'both')
>>> mla.lockin.set_trigger_flanks(2, 'positive')
```

set_trigger_in_holdoff(*trig_in_port*, *hold_off_time_seconds*)

Set a time after a trigger event during which new trigger events cannot be registered.

Parameters

- trig_in_port – 1 or 2
- hold_off_time_seconds – float [seconds]

Note: This is useful to avoid false trigger events in noisy or bouncing signals.

start_lockin(*dma_channel=1*, *trigger_mode=1*, *cluster_size='auto'*)

Tell the MLA to start streaming lockin data packets.

Parameters

- dma_channel – 0 or 1. In standard amplications time data use dma_channel=0 and lockin data use dma_channel=1
- trigger_mode – 1 - start immediately or 2 - start on trigger event on trigger in 1.
- cluster_size – integer > 0, or auto. Set how many pixels should be combined into a cluseter before it is transmitted.

Returns

None

Note: The lockin calculations (and feedback) are always running. This function only activates the ethernet transmission. Normally, you don't run this function with trigger_mode=2. Run arm_trigger() instead.

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.stop_lockin()
```

See also:

stop_lockin(), arm_trigger()

start_parameter_sweep(*df_array=None*, *f_array=None*, *phase_array=None*, *amp_array=None*, *pixels_per_step=1*, *phase_unit='rad'*, *amp_unit='calibrated'*)

Start parameter sweep, native / digital input units

Parameters

- df_array – np.array shape (pts,), measurement bandwidth for each step in sweep
- f_array – 2D np.array shape (n_tones, pts), frequency values
- shape (phase_array 2D np.array) –
- amp_array – 2D np.array shape (n_tones, pts), drive amplitude for each step in sweep
- pixels_per_step – int, wait for this many pixels at each parameter step
- phase_unit – str, 'rad' (default) or 'degree'
- amp_unit – str, 'calibrated' (default), 'percent', or 'digital'

Returns
None

Details:
This function uses hardware-enabled functionality for a performing a fast parameter sweep without having to realy on communication between the computer and the MLA between each step in the sweep. This function only starts the sweep. Use get_parameter_sweep() to read out the results. Any parameter can either have value None (not swept) or be a np.array of parameter values to be swept. All parameters must arrays must have the same number of pts, but do not need the same number of tones (n_tones)

start_parameter_sweep_native(*spp_array=None*, *pa_array=None*, *phase_array=None*, *amp_array=None*, *pixels_per_step=1*)

Start parameter sweep, native / digital input units

Parameters

- spp_array – np.array, samples per pixel for each step in sweep
- pa_array – 2D np.array shape (n_tones, pts), phase accumulator values for each step in sweep
- shape (phase_array 2D np.array) –
- amp_array – 2D np.array shape (n_tones, pts), drive amplitude for each step in sweep
- pixels_per_step – int, wait for this many pixels at each parameter step

Returns
None

Details:
This function uses hardware-enabled functionality for a performing a fast parameter sweep without having to realy on communication between the computer and the MLA between each step in the sweep. This function only starts the sweep. Use get_frequency_sweep() to read out the results. Any parameter can either have value None (not swept) or be a np.array of parameter values to be swept. All parameters must arrays must have the same length (first dimension). Some parameters

such as pa, phase and amp are themselves arrays with the frequency index as the second dimension. Second dimensions do not need to be equal.

stop_lockin(*dma_channel=1*, *force=False*)

Tell the MLA to stop streaming lockin data packets.

Parameters

- dma_channel – 0 or 1. In standard amplications time data use dma_channel=0 and lockin data use dma_channel=1
- force – boolean. Force the stream to stop even if there are remaining lockin recievers.

Returns
None

Note: The lockin calculations (and feedback) are always running. This function only deactivates the ethernet transmission.

Examples

```
>>> mla.lockin.stop_lockin()
```

See also:

start_lockin(), arm_trigger()

tune0(*f*, *df*, *native_output_units=False*)

Performs only the minumum amount of tuning necessary to make the requested numbers fit the corresponding digtal registers in the MLA hardware.

Parameters

- f (float or np.ndarray) – target frequencies in Hz.
- df (float) – target measurement bandwidth in Hz.
- native_output_units (bool, optional) – return f and df in digital units rather than Hertz.

Returns
The tuned frequencies in Hz. df_tuned (float): The tuned measurment bandwidth in Hz.

Return type
f_tuned (float or np.ndarray)

Note: This tuning does not prevent Fourier leakage. This function is only to be used in rare cases where Fourier leakage is known not to be an issue. For example when measuring strong tones well separated in frequency. If you are using this function, you will probably want to also use *mla.lockin.reset_tones_on_new_pixel(False)* to avoid jumps in the signal at the end of each measurement time window.

Examples

```
>>> f_out, df_out = mla.lockin.tune0(123456.789, 1000)
>>> mla.lockin.reset_tones_on_new_pixel(False)
>>> mla.lockin.set_frequencies(f_out)
>>> mla.lockin.set_df(df_out)
```

See also:

tune1() tune2()

tune1(*f*, *df*, *priority='f'*, *native_output_units=False*, *rounding=1*, *regular=False*)

Performs standard level of tuning, adequate for most measurements.

Parameters

- f (float or np.ndarray) – target frequencies in Hz.
- df (float) – target measurement bandwidth in Hz.
- priority (str, optional) – tuning priority, see Usage.
- native_output_units (bool, optional) – return f and df in digital units rather than Hertz.
- rounding (int, optional) – if set to m > 1, it will ensure Fourier leakage is minimized also in case of using a downsampling of m
- regular (bool, optional) – require that the number of samples per pixel is a regular number, see Notes.

Returns

The tuned frequencies in Hz. df_tuned (float): The tuned measurment bandwidth in Hz.

Return type

f_tuned (float or np.ndarray)

Usage:

The returned bandwidth and frequencies are tuned so that they fulfill the condition f_i = n_i*df which avoids Fourier leakage. With priority 'f' the highest priority is put on getting the frequencies as close as possible to the target frequencies. With priority 'df' the highest priority is put on getting the measurement bandwidth as close as possible to the target value. This tuning greatly supresses Fourier leakage, but does not completely prevent it. This function should be used with *mla.lockin.reset_tones_on_new_pixel(True)*, which is the default state. When the reset function is activated, Fourier leakage will be immeasurable for any reasonalbe bandwidth.

Notes

If the lockin is used in parallel with a custom FFT calculation, setting regular=True ensures that the number of samples per pixel spp has only 2, 3 or 5 as prime factors (spp is 5-smooth, or a regular number). This can significantly speed up the FFT.

Examples

```
>>> f_out, df_out = mla.lockin.tune1(123456.789, 1000)
>>> mla.lockin.set_frequencies(f_out)
>>> mla.lockin.set_df(df_out)
```

See also:

tune0() tune2()

tune2(*f*, *df*, *priority='f'*, *native_output_units=False*)

Performs the most rigid tuning. With this tuning there will be zero Fourier leakage and very stable phase, but it is more limited in the possible values of the measurement bandwidth.

Parameters

- f (float or np.ndarray) – target frequencies in Hz.
- df (float) – target measurement bandwidth in Hz.
- priority (str, optional) – tuning priority, see Usage.
- native_output_units (bool, optional) – return f and df in digital units rather than Hertz.

Returns

The tuned frequencies in Hz. df_tuned (float): The tuned measurment bandwidth in Hz.

Return type

f_tuned (float or np.ndarray)

Usage:

The returned bandwidth and frequencies are tuned so that they fulfill the condition f_i = n_i*df which avoids Fourier leakage. With priority 'f' the highest priority is put on getting the frequencies as close as possible to the target frequencies. With priority 'df' the highest priority is put on getting the measurement bandwidth as close as possible to the target value. With either prioity, the bandwidth is chosen such that the number of samples in the measurment time window is an integer power of 2. This tuning is recomended when the MLA is snychrnoized to external generators or sampling cards.

Examples

```
>>> f_out, df_out = mla.lockin.tune2(123456.789, 1000)
>>> mla.lockin.set_frequencies(f_out)
>>> mla.lockin.set_df(df_out)
```

See also:

tune0() tune1()

wait_for_counter(*counter*, *increment=1*, *target_value=None*)

Block computer execution until specified counter in the received lockin data has reached a specified falue.

Parameters

- counter – interger or string, must be one of the following:

  1 = 'settings' (increments every time a parameter is
  changed. This counter can be used to identify which data had which parameters. Note that the frequency and phase parameters will increment the settings counter by 2 when they are set.)

2 = 'trig2' = 'image_nr' (count the number of trigger
events in TRIG IN 2)

3 = 'trig1' = 'line_nr' (count the number of trigger
events in TRIG IN 1)

4 = 'pixel' counter for the total number of time windows
that has passed since the MLA was powered on. Note that this counter will continue to increase regardless of weather the data was transmitted to the computer or not.

- increment – integer, wait for the specified counter to increment by this value. Can be overridden by target_value.
- target_value – integer or None. If integer, override the increment parameter and wait for the specified counter to reach this value.

Returns
buffer position of the first lockin packet which contains the specified counter and value.

Return type
integer

Note: The lockin must be running, otherwise no new data will reach the computer data buffer. To continue the execution without having receiving the specified counter you can run abort_wait_for_counter() from another thread.

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.wait_for_counter('pixel', increment=10)
1892
>>> mla.lockin.stop_lockin()
```

See also:

abort_wait_for_counter(), wait_for_new_pixels(), wait_for_trigger()

wait_for_new_pixels(*n=1*, *timeout=9999*)

Block computer execution until the specified number of new pixels has been received in the computer buffer.

Parameters

- n – integer, then number of new pixels to wait for
- timeout – float or 'auto', raise TimeOutException after this time if not enough pixels have arrived.

Returns
None

Note: The lockin must be running, otherwise no new data will reach the computer data buffer. To continue the execution without having receiving the specified counter you can run abort_wait_for_new_pixels() from another thread.

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.wait_for_new_pixels(10)
>>> mla.lockin.stop_lockin()
```

See also:

abort_wait_for_new_pixels(), wait_for_counter(), wait_for_trigger()

wait_for_settings_effect(*timeout=None*)

When a command that changes the setup of the lockin is executed, there will inevitably be a delay until the change has had an effect in the hardware. After the new settings have had an effect in the hardware, and new data has been measured with the new settings, there will another delay until that data is available to the user. This function blocks further execution until lockin data with settings counter cnt are available.

Parameters
timeout – float [seconds], return after this time, even if the setting was not

Returns
None

Note: This function is automatically called when a setting is changed unless the keyword wait_for_effect=False is specified when the setting is changed. The purpose if this function is to allow the user to speed up their measurment scripts that changes multiple settings in each iteration of a loop, see the example below.

Examples

```
>>> mla.lockin.start_lockin()
>>> mla.lockin.set_frequencies(100e3, wait_for_effect=False)
>>> mla.lockin.set_amplitudes(0.1, wait_for_effect=False)
>>> mla.lockin.set_Tm(0.01, wait_for_effect=False)
>>> mla.lockin.wait_for_settings_effect()
>>> p = mla.lockin.get_pixels(n_pix=1, data_format='IQreal', unit='digital')
>>> p
array([ 2.20439491e+02, -1.00917580e+02, ..., -1.47609380e+02])
>>> mla.lockin.stop_lockin()
```

wait_for_trigger()

Block computer execution until received lockin data shows incremented trigger counter. This function will only work for the port TRIG IN 1. Please contact Intermodulation Products if you need other trigger methods.

Returns
buffer position of the lockin packet with increment trigger counter.

Return type
integer

Note: The line-pipe must be enabled before you can use this function. The lockin must be running, otherwise no new data will reach the computer data buffer. To continue the execution without receiving a trigger you can run abort_wait_for_trigger() from another thread.

Examples

```
>>> from threading import Timer
>>> mla.lockin.set_trigger_flanks(trig1_flank='positive', trig2_flank='positive')
>>> mla.lockin.enable_wait_for_trigger(True)
>>> mla.lockin.start_lockin()
>>> Timer(1, mla.lockin.abort_wait_for_trigger).start()
>>> bufpos = mla.lockin.wait_for_trigger()
>>> mla.lockin.trig2_cnt[bufpos]
37
>>> mla.lockin.stop_lockin()
```

See also:

enable_wait_for_trigger(), set_trigger_flanks(), abort_wait_for_trigger()

class mlaapi.lockin.LockinReceiver(*lockin*)

A LockinReceiver is a separate object for extracting lock-in data from the buffer. Each LockinReceiver keeps track of which data it has extracted from the buffer. Use it when you have multiple, threaded, measurements running at the same time.

get_new_pixels(*data_format='complex'*, *unit='calibrated'*)

Return all new lockin packets from the buffer. The lockin object keeps track of which data has been extracted from the buffer so that this function only returns new data that was not extraced last time. This command is usually issued in a loop together with plotting and other data processing commands. Before the loop the start of the measurement is indicated with the command *mla.lockin.get_new_pixels(throw_away=True)* which will throw away all the data that has arrived previous to this command.

Parameters

- data_format ('IQreal', 'complex', 'amp', 'phase') – specifies return data type.
- unit ('calibrated', 'digital', 'deg, 'rad') – specifies units. ‘rad’ and ‘deg’ only for phase.

Returns

pixels, nr_pixels, meta

- pixels : np.array(dtype=float) [ADU], axis 0 = [f0, f1, f2,. . .], axis 1 = [pixel0, pixel1, pixel2]
- nr_pixels (integer): number of pixels returned
- meta (Two dimensional numpy array of integers): The meta data of the pixels, i.e. header, settings_counter, trig2 counter, trig1 counter, pixel counter, trigger position, dc_data.

Return type

Tuple

Examples

```
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.start_lockin()
>>> rec = lockin.LockinReceiver(mla.lockin)
>>> rec.wait_for_new_pixels(10)
>>> p, n, m = rec.get_new_pixels()
>>> p.shape
(42L, 10L)
>>> n
10
>>> m.shape
(6L, 10L)
>>> mla.lockin.stop_lockin()
```

See also:

mlaapi.lockin.Lockin.get_tone(), mlaapi.lockin.Lockin.get_line(), mlaapi.lockin.Lockin.get_pixel_average(), mlaapi.lockin.Lockin.get_pixels(), *Calibration*

wait_for_new_pixels(*n=1*, *timeout=100*)

Block computer execution until the specified number of pixels has been received since the last call to get_new_pixels().

Parameters

- n – integer, then number of new pixels to wait for
- timeout – float or 'auto', raise TimeOutException after this time if not enough pixels have arrived.

Returns

None

Note: The lockin must be running, otherwise no new data will reach the computer data buffer. To continue the execution without having receiving the specified counter you can run mlaapi.lockin.Lockin.abort_wait_for_new_pixels() from another thread.

Examples

```
>>> mla.lockin.set_Tm(0.1)
>>> mla.lockin.start_lockin()
>>> rec = lockin.LockinReceiver(mla.lockin)
>>> rec.wait_for_new_pixels(10)
>>> p, n, m = rec.get_new_pixels()
>>> p.shape
(42L, 10L)
>>> n
10
>>> m.shape
(6L, 10L)
>>> mla.lockin.stop_lockin()
```

See also:

mlaapi.lockin.Lockin.abort_wait_for_new_pixels(), mlaapi.lockin.Lockin.wait_for_counter(), mlaapi.lockin.Lockin.wait_for_trigger()

7.2.3 mla.osc

class mlaapi.osc.Osc(*hardware*)

The mla.osc object deals with sampling of time domain data. Just as with the lockin object, starting a measurement is a nonblocking command that automatically starts to fill a computer buffer. Reading out measurement data is done by a separate command.

The mla.osc object also has functions for setting up downsampling and other functions related to acquiring time domain data.

ad_is_in_safe_range(*nr_samples*, *port=[1]*)

Make a number of samples with the AD converter and report weather all samples is within 96% of the AD converter range.

Parameters

- nr_samples (int) – number of samples to use for the test.
- port (int or list, optional) – {1-4} select which physical port to use for the measurement. Can be a list.

Returns

if True, all samples are within 96% of the AD converter range.

Return type

bool

Examples

```
>>> mla.osc.ad_is_in_safe_range(mla.lockin.samples_per_pixel)
True
```

extract_triggers(*data_raw*)

Separate sample data from triggers when using interleave_eol=True in start_time_stream

Parameters

data_raw (numpy array of int16) – The raw time stream data with interleaved triggers

Returns

sample data list: triggers

Return type

numpy array of int16

Example

```
>>> mla.osc.start_time_stream(1e5, in_port=1, interleave_eol=True)
>>> data_raw = mla.osc.get_data(1e5)
>>> data_clean, triggers = extract_triggers(data_raw)
>>> data_V = mla.convert.AD_to_V(data_clean, 1)
```

See also:

start_time_stream(), get_data()

frequency_count(*nsamples=0*, *channel=1*, *downsampling=1*)

Acquire a number of time samples and return the frequency of the strongest tone.

Parameters

- nsamples (integer) – The number of samples to acquire.
- channel (integer) The physical port (1-4) –

- downsampling (integer) – Set a downsampling factor before acquiring the data

Returns
The frequency of the strongest tone.

Return type
float [Hz]

Examples

```
>>> mla.lockin.set_output_mask(1)
>>> mla.lockin.set_amplitudes(0.1)
>>> mla.lockin.set_frequencies_khz(123.456789)
>>> mla.lockin.set_df(100)
>>> mla.lockin.reset_tones_on_new_pixel(False)
>>> time.sleep(1)
>>> mla.osc.frequency_count(1e6)
123456.78...
```

get_data(*nr_samples*, *copy_data=True*, *timeout=None*)

Extract sample data from the computer buffer. Use this command when acquiring single bursts of data, i.e. together with the restart_when_done=False keyword.

Parameters

- nr_samples (integer) – The number of samples to extract
- copy_data (boolean) – If True return a copy of the data, otherwise return pointer to shared buffer.
- timeout (float, optional) – maximum time to wait for samples in milliseconds. Set to None (default) to let the oscilloscope calculate the timeout. Set to 0. to wait forever.

Returns
The raw samples.

Return type
numpy array of int16

Raises
OscTimeoutError – if no data is available within timeout milliseconds.

Example

```
>>> mla.osc.start_time_stream(1e6)
>>> data = mla.osc.get_data(1e6)
>>> data
array([-300, -264, -260, ..., -324, -280, -268], dtype=int16)
>>> data.shape
(1000000,)
```

See also:

start_time_stream(), get_stream_data()

get_max_DMA_rate()

Return the maximum rate of internal memory transfer on the FPGA. The total data rate (nr ports * sample rate) can not exceed this number when running time_stream

Returns
samples per second

Return type
float [Hz]

get_max_transfer_rate()

Return the approximate maximum rate of transfer data from MLA to computer Sets the maximum rate of continous data transer. For single shot measurements higher rates can be used and the data will be buffered on the MLA.

Returns
samples per second

Return type
float [Hz]

get_n_samples_max(*dma_channel=0*)

Return the maximum number of time samples that fits into the MLA RAM memory in the current configuration. The MLA RAM memory can be reconfigured with mlaapi.hardware.Hardware.configure_memory().

Parameters
1] (dma_channel [0 or) – Selects which DMA channel to query.

Returns
The maximum number of time samples that fits into the MLA RAM memory in the current configuration.

Return type
integer

Example

```
>>> M = 1<<20 # One Megabyte
>>> mla.hardware.configure_memory(M, M, M, M, 760*M, M, M, M)
>>> mla.osc.get_n_samples_max(0)
398458880
>>> mla.osc.get_n_samples_max(1)
524288
>>> mla.hardware.set_default_settings()
>>> mla.osc.get_n_samples_max(0)
67108864
>>> mla.osc.get_n_samples_max(1)
67108864
```

See also:

start_time_stream(), mlaapi.hardware.Hardware.configure_memory()

get_samplerate()

Return the sampling frequency of the fast AD converters, taking specified downsampling into account.

Returns
The sampling frequency.

Return type
float [Hz]

Note: The downsampling is associated with the DMA channel for the data transfer. Not the physical channel.

Examples

```
>>> mla.osc.set_downsampling(16)
>>> mla.osc.get_samplerate()
3125000.0
>>> mla.osc.set_downsampling(1)
>>> mla.osc.get_samplerate()
50000000.0
```

See also:

set_downsampling(), get_samplerate_raw()

get_samplerate_raw()

Return the raw sampling frequency of the fast AD converters, not taking any downsampling into account.

Returns
The sampling frequency.

Return type
float [Hz]

Note: The downsampling is associated with the DMA channel for the data transfer. Not the physical channel.

Examples

```
>>> mla.hardware.set_ad_clock_divisor(25)
>>> mla.osc.get_samplerate_raw()
100000000.0
>>> mla.hardware.set_ad_clock_divisor(50)
>>> mla.osc.get_samplerate_raw()
50000000.0
```

See also:

set_downsampling(), get_samplerate_raw()

get_stream_data(*nr_samples*, *timeout=None*)

Extract sample data from the computer buffer. Use this command together when acquiring data continously, i.e. together with the restart_when_done=True keyword. The idea is to use this command repeatingly: acqiure one chunks, process the data, and then acquire a new chunk. The function automatically make sure that the next chunk is

Parameters

- nr_samples (int) – The number of samples to extract.
- timeout (float, optional) – maximum time to wait for samples in milliseconds. Seto to None (default) to let the oscilloscope calculate the timeout. Set to 0. to wait forever.

Returns
The raw samples.

Return type
numpy array of int16

Raises
OscTimeoutError – if no data is available within timeout milliseconds.

Example

```
>>> # Extract the maximum of one million samples
>>> m = -32768
>>> mla.osc.start_time_stream(1e5, restart_when_done=True)
>>> for i in range(10):
...     data = mla.osc.get_stream_data(1e5)
...     m = max(m, data.max())
>>> m
3134
>>> mla.osc.stop_time_stream()
```

See also:

start_time_stream(), get_data()

set_downsampling(*n=1*, *multiplier=None*)

Setup downsampling in the FPGA logic. The downsampling algorithm adds n samples and multiplies the sum by a multiplier. The lowest 16 bits is then used as the output. This means multiplier=2**16 corresponds to scaling the output by 1.

Normally the multiplier can be left to None, which meas the it will be calculated automatically to preserve the magnitue of the output regardless of the number of samples, (i.e. multiplier = 2**16/n). The only reason to set the multiplier manually is when you are sure that the input signal is very low, and you want to average many samples without throwing away two least significant bits.

Parameters

- dma_channel – 0 or 1: select which DMA channel to activate downsampling on.
- n – integer < 255: how many samples should be used in the downsampling
- multiplier – integer < 2**16

Note: The downsampling is associated with the DMA channel for the data transfer. Not the physical channel.

Examples

```
>>> mla.osc.set_downsampling(16)
>>> mla.osc.start_time_stream(25e5, trig_mode=1)
>>> t0 = time.time()
>>> data = mla.osc.get_data(25e5)
>>> time.time()-t0
0.8
>>> mla.osc.set_downsampling(1)
>>> mla.osc.start_time_stream(25e5, trig_mode=1)
>>> t0 = time.time()
>>> data = mla.osc.get_data(25e5)
>>> time.time()-t0
0.05
```

See also:

get_samplerate()

set_trigmode(*trig_mode*, *osc=True*, *arb=False*)

Set trigger mode for an already started time stream.

This function can be used to simulataneously trigger both sampling ("osc") and arbitrary waveform ("arb").

Parameters

- trig_mode (integer) – Select how to trigger the start of the data stream. See table below
- osc (boolean) – Set trigger mode for mla.osc.
- arb (boolean) – Set trigger mode for mla.arb to trigger simultaneously with osc time stream.

trig_mode	Description
0	Not triggered (the stream will not start)
1	Triggered (the stream starts immediately)
2	Trigger when the lock-in unit starts a new measurement window
3	Trigger when trigger in 1 event has arrived, and the lock-in unit starts a new measurement window

See also:

mlaapi.arb.Arb.set_trigmode()

start_time_stream(*nr_samples*, *reset=True*, *dma_channel=0*, *restart_when_done=False*, *trig_mode=2*, *port=None*, *interleave_eol=False*)

Start sending raw sample data from the AD converters to the computer buffer.

Parameters

- nr_samples (integer) – The number of samples you want to acquire.
- reset (boolean) – Deprecated, no effect
- dma_channel (integer) – 0 or 1. Select which DMA channel to use to transfer the data from the FPGA logic to the MLA RAM memory. Normally, channel 0 is used for time samples and channel 1 is used for lockin data.
- restart_when_done (boolean) – Tells the MLA to automatically start a new transfer immediately when the previous transfer is finished. Use this command to transfer streams that are longer than what fits into the MLA RAM (Typically 64 Msamples, but that can be reconfigured, see mlaapi.hardware.Hardware.configure_memory()).
- trig_mode (integer) – Select how to trigger the start of the data stream. See table below
- port (integer or list) – Select which physical port [1-4] to use for the measurement. Can be a list.
- interleave_eol (boolean) – Interleave trigger in 1 events with data stream. When active, use extract_triggers() to separate sample data and triggers.

trig_mode	Description
0	Not triggered (the stream will not start)
1	Triggered (the stream starts immediately)
2	Trigger when the lock-in unit starts a new measurement window
3	Trigger when trigger in 1 event has arrived, and the lock-in unit starts a new measurement window

Examples

```
>>> mla.osc.start_time_stream(50000)
>>> mla.osc.get_data(50000)
array([-300, -260, -304, ..., -268, -272, -296], dtype=int16)
```

See also:

get_n_samples_max(), get_data(), get_stream_data()

stop_time_stream(*dma_channel=0*)
Stop continous streaming of time data.

Examples

```
>>> mla.osc.start_time_stream(1e5, restart_when_done=True)
>>> data = mla.osc.get_stream_data(1e6)
>>> data
array([-280, -296, -284, ..., -296, -288, -288], dtype=int16)
>>> data.shape
(1000000,)
>>> mla.osc.stop_time_stream()
```

7.2.4 mla.analog

class mlaapi.analog.Analog(*mla*)

This class is used to set up the the analog front end of the MLA board. Here, you can control gain, input impedance and AC/DC-coupling.

Here below, only port IN 1 is described. But all four input ports are have the same set of switches, and all ports can be controlled individually.

What is the default range.

Discription of the signal path from the SMA connector, to the AD-converter:

1. The SE/DIFF switch determines weather to use the port in single ended mode (i.e. use only IN 1+), or in differential mode (i.e. both IN 1+ and IN 1-) If SE-mode is chosen IN 1- will be grounded.
2. The INDIV/SHORT switch
3. The HIGH_Z/50 switch determines the input impedance.
4. The AC/DC switch determines weather or not to an AC-coupling capacitor of 0.1 uH.
5. The GAIN1/GAIN5 switch sets the gain of the first stage preamplifier.

The standard user will use the preamplifier (PREAMP) in which case the bahavior of the channel is easy to predict:

Advanced user may want to optimize the switch configuration for their particular experiment. First deter

1. **You need to use the PREAMP if you fulfil any of the follwing:**
   - You need DC-coupling _and_ you do not have your own pre-amplifier that can handle a DC-offset on the IN-port of the MLA.
   - You need more than 2 kOhm input impedance at gain=0.3
   - You need more than 1 kOhm inpur impedance at gain=0.5
   - You need more than 100 Ohm input impedace at gain=6

- You need more than gain=6.

If you don't need the preamp, you may get better distorsion performance in BYPASS mode.

BYPASS MODE Gain Single ended Input impedance Differential imput impedance 0.29 2.1 kOHm or 47 0.55 1.1 kOHm or 47 6 100 or 33 0.033 1.03 kHom

1. Do you need DC-coupling? In this set choose AC/DC to DC (duh) and set PREAMP/BYPASS to PREAMP (if you don't use the PREAMP, a DC voltage from the AD-converter will propagate to the input connector. If you have your own preamplifier that could handle this DC voltage, you can still use BYPASS mode).
2. Do you need 50 Ohm termination? Set HIGH_Z/50 to 50.

Describe the circuit. Name all switches.

Describe wait_for_effect.

get_input_range(*in_port*)

Get the input voltage range

Parameters
in_port (int) – [1-4] determines which port to set

Returns
range_volt (float)

See also:

set_input_range()

get_input_range_selection(*in_port*)

Get the current input voltage range selection

Parameters
in_port (int) – [1-4] determines which port to set

Returns
range_selection (int)

See also:

set_input_range() get_input_ranges()

get_output_range(*out_port*)

Get the output voltage range

Parameters
out_port (int) – [1-2] determines which port to set

Returns
range_volt (float)

See also:

set_output_range() get_output_ranges()

get_output_range_selection(*out_port*)

Get the output voltage range selection

Parameters
out_port (int) – [1-2] determines which port to set

Returns
range_selection (int)

See also:

set_output_range() get_output_ranges()

get_output_ranges(*out_port*)

Get the available output ranges in calibrated units

Parameters
out_port (int) – [1-2] determines which port to

Returns
list of voltage ranges for the different selection

Return type
range_selections (list)

See also:

set_output_range() get_output_range()

is_valid_input_range(*input_range: InputRange*)

Check if input_range is a valid setting for this MLA

set_dc_coupling(*in_port*, *dc_coupling*)

Set the in_port to be AC or DC coupled.

Parameters

- in_port (int) – [1-4] determines which port to set
- dc_coupling (bool) – True means DC coupled, False means AC-coupled

Returns
None

The AC-coupling capacitance of 0.1 uF will constitute a high-pass RC-filter together with the following resistance. When the first stage preamplifier is used, the resistance is 1 MOhm, which means that the cut-off frequency is 1.6 Hz. If the first stage preamplifier is bypassed, the resistance will depend on the GAIN2 switch.

Notes

AC-coupling will automatically be enforced if a large voltage that could damage the circuits is applied to the input.

See also:

set_bypass()

References

Put references, web links, or links to other sections of the user manual here.

set_input_range(*in_port: int*, *input_range: InputRange*)

Set the input voltage range

Parameters

- in_port (int) – [1-4] determines which port to set
- input_range (InputRange) – one of mlaapi.analog.InputRange

Returns
None

See also:

get_output_range() get_output_ranges()

set_input_single_ended(*in_port: int*, *single_ended: bool = True*)

Set single-ended or differential input

Parameters

- in_port (int) – [1-4] input port
- single_ended (bool) – True - single ended, False - differential

set_input_termination_50ohm(*in_port: int*, *term: bool = True*)

Set input termination

Parameters

- in_port (int) – [1-4] input port
- term (bool) – False - high impedance, True - 50 ohm termination

set_output_range(*out_port: int*, *range_selection: OutputRange*)

Set the output voltage range

Parameters

- out_port (int) – [1-2] determines which port to set
- range_selection (OutputRange) – low (0) or high (1) range, see get_output_ranges for range values

Returns

None

See also:

get_output_range() get_output_ranges()

7.2.5 mla.arb

class mlaapi.arb.Arb(*hardware*)

extend_waveform(*waveform*, *repeat*)

Helper function to extend waveform so that number of samples is divisible by 64

Depending on the repeat argument the waveform is either zero-padded (repeat=False) or repeated in memory (repeat=True).

Parameters

- [ndarray] (waveform) –
- False] (repeat [True or) –

Return type

extended waveform

get_max_nsamples(*port=1*)

Return the max length accepted for an arbitrary waveform

By default equal amount of memory is assigned to each output channel and each input channel. Max number of samples can be increased by changing memory allocation.

Parameters

port (int, optional) – Output port number. The default is 1.

Returns

Max number of samples allowed in waveform at specific port.

Return type

int

set_pulse(*freq*, *ampl*, *width*, *port=1*, *nsamples=1000000*, *unit='calibrated'*, *correct_delay=True*)

Set a square pulse with given width.

Parameters

- freq – pulse repetition frequency
- ampl – pulse height
- width – pulse width relative to repetition period
- port – output port, 1 or 2
- nsamples – number of samples to generate
- delay – shift the waveform to the right (left) if delay > 0 (< 0)

set_pulse2(*freq*, *ampl*, *offset=0*, *width=0.5*, *rise=0.0*, *fall=0.0*, *port=1*, *nsamples=1000000*, *unit='calibrated'*, *correct_delay=True*)

Set a square pulse with given parameters.

Parameters

- freq (float) – pulse repetition frequency in Hz
- ampl (int) – pulse amplitude in DA (digital) units
- offset (int, optional) – pulse low value in DA (digital) units
- width (float) – pulse width in units of repetition period in the open interval (0, 1)
- rise (float, optional) – pulse rise time constant in units of repetition period [0, inf)
- fall (float, optional) – pulse fall time constant in units of repetition period [0, inf)
- port (int, optional) – output port, 1 or 2
- nsamples (int, optional) – number of samples to generate
- delay (bool, optional) – correct for delay in the DAC

set_trigmode(*trig_mode*, *osc=False*, *arb=True*)

Set trigger mode for an already started arb.

This function can be used to simulataneously trigger both sampling ("osc") and arbitrary waveform ("arb").

Parameters

- [integer] (trig_mode) – Select how to trigger the start of the data stream. See table below
- [boolean] (arb) – Set trigger mode for mla.osc.
- [boolean] – Set trigger mode for mla.arb to trigger simultaneously with osc time stream.

trig_mode	Description
0	Not triggered (the stream will not start)
1	Triggered (the stream starts immediately)
2	Trigger when the lock-in unit starts a new measurement window
3	Trigger when trigger in 1 event has arrived, and the lock-in unit starts a new measurement window

See also:

mlaapi.osc.Osc.set_trigmode()

set_waveform(*waveform: ndarray*, *port=1*, *unit='calibrated'*, *correct_delay=True*)

Set arbitrary waveform data

Parameters

- waveform (np.array float) – Arbitrary waveform data. Length must be divisible by 64. See extend_waveform() for a helper-function to fix waveform length.
- port (int, optional) – Output port, 1 or 2. The default is 1.
- unit ({'calibrated', 'digital', 'percent' , '%'}, optional) – Unit of waveform data. 'calibrated' typically means Volt. The default is 'calibrated'.
- correct_delay (TYPE, optional) – Apply a correction for the DAC delay by circulatory shifting the data. Should be applied only for repeating signals. The default is True.

Raises

ValueError –

Return type

None.

start_arb(*port=None*, *trig_mode=2*, *repeat=True*)

Start ARB.

Parameters

- int] (port [tuple of) – list of ports e.g. [1] (default), [2] or [1, 2] for both ports.
- [integer] (trig_mode) – Select how to trigger the start of the data stream. See table below
- False] (repeat [True or) – Restart the wavefrom from the beginning when it reaches the end

trig_mode	Description
0	Not triggered (the stream will not start)
1	Triggered (the stream starts immediately)
2	Trigger when the lock-in unit starts a new measurement window
3	Trigger when trigger in 1 event has arrived, and the lock-in unit starts a new measurement window

Note: You must use mla.hardware.set_output_type("arb", port) before using this function

stop_arb()

Stop arb and set output type to "lockin"

Set the fpga output multiplexer to output data from the lockin waveform generators, as opposed to the arbitrary waveform generators.

7.2.6 mla.feeback

class mlaapi.feedback.Feedback(*hardware*, *lockin*, *osc*, *settings*)

activate_phase(*enable=True*)

Activates or deactivates phase feedback on port OUT C.

Parameters

enable – True or False

See also:

set_phase_offset(), set_current_phase_to_value()

afm_engage(*relative_tapping_mode_setpoint=None*, *event=True*)

Reenable the feedback.

Parameters

relative_tapping_mode_setpoint (float, optional) – if missing, uses the last used.

Notes

sets the feedback mode and adjusts the setpoint.

See also:

afm_lift

afm_lift(*event=False*)

Switch off the feedback and cause the AFM to lift.

Notes

Sets the feedback mode to dc_only and outputs a big dc offset.

See also:

afm_engage

autosetup_afm(*relative_tapping_mode_setpoint*, *verbose=True*)

Set up the amplitude feedback and enable it on the correct output port.

Parameters

- relative_tapping_mode_setpoint (float) – the output will be zero when the measured amplitude is the current amplitude times this factor
- verbose (bool, optional) – set to False to avoid printing to stdout

Notes

The output port is chosen according to settings['FEEDBACK']['fb_out_port'] The feedback type is chosen according to settings['FEEDBACK']['polarity']

See also:

set_feedback_type_on_correct_port

calculate_feedback_amplitude(*pix*, *verbose=True*)

(Re)calculates the cordic contribution to the feedback amplitude at free amplitude.

Parameters

pix (ndarray) – a scaled lockin pixel in IQreal format and digital units

Returns

the calculated cordic feedback amplitude

Return type

(int)

Notes

Reads class attributes _cordic_shift and _cordic_scale. Sets class attribute _feedback_amplitude

set_current_phase_to_value(*value=0*, *unit='calibrated'*)

Make the current reading of the phase generate a specific value at OUT C.

Parameters

- value – float, The value that the current reading of phase should generate at OUT C
- unit – 'calibrated' or 'digital'

See also:

activate_phase(), set_phase_offset(), *Calibration*

set_feedback_type(*val*)

0 feedback from ublaze 1 only dc value 2 cordic feedback, legacy mode 3 cordic feedback, positive photodiode polarity 4 cordic feedback, negative photodiode polarity 5 Use feedback port as a signal output port 7 Output In-phase part 8 Output Quadrature-phase part

set_feedback_type_on_correct_port(*val*)

0 feedback from ublaze 1 only dc value 2 cordic feedback, legacy mode 3 cordic feedback, positive photodiode polarity 4 cordic feedback, negative photodiode polarity 5 Use feedback port as a signal output port 7 In-phase part (and Quadrature-phase part on OUT C if fb_out_port is slow_dac) 8 Quadrature-phase part (only on OUT 2)

set_feedback_type_on_port(*val*, *out_port*)

0 feedback from ublaze 1 only dc value 2 cordic feedback, legacy mode 3 cordic feedback, positive photodiode polarity 4 cordic feedback, negative photodiode polarity 5 Use feedback port as a signal output port 7 In-phase part (and Quadrature-phase part on OUT C if out_port is slow_dac) 8 Quadrature-phase part

set_feedback_type_slow(*val*)

0 feedback from ublaze 1 only dc value 2 cordic feedback, legacy mode 3 cordic feedback, positive photodiode polarity 4 cordic feedback, negative photodiode polarity 5 Use feedback port as a signal output port 7 Output In-phase part (OUT A) and Quadrature-phase part (OUT C) instead of amplitude and phase

set_phase_offset(*offset*, *unit='calibrated'*)

Add a constant offset to the phase feedback value to port OUT C.

Parameters

- offset – 16 bit integer, -32768 to 32767 if unit is digital
- unit – 'calibrated' or 'digital'

Note: This function is useful, both for setting a custom set point, and for avoiding jumps when the phase is wrapping.

See also:

activate_phase(), set_current_phase_to_value()

setup(*gain=1.0*, *offset=0.0*, *output_port='A'*, *unit='calibrated'*)

Set up an analog output voltage, proportional to the result of a lock-in measurment using the frequency at index=0.

Parameters

- gain (float, optional) – With gain=1.0 the output voltage will be equal the amplitude of the input voltage.

- offset (float, optional) – Adds an offset to the output voltage.
- output_port (int or str, optional) – {2, 'A', 'both'} specify which port shall be used for the analog output. Default output port A.
- unit (str, optional) – {'calibrated', 'digital'} specify which unit to work with. When 'calibrated', use the units of the selected calibration. When 'digital', use the direct digital units of the AD- and DA-converters.

Returns
whether the given combunation of parameters were valid.

Return type
(bool)

Raises
ValueError – if the values of the arguments are not supported.

Notes

Assuming the input signal is A_in * cos(w*t), setting gain=1.0 will produce an output equal to the measured (peak) amplitude A_in. For the peak-to-peak amplitude, set gain=2.0 For the RMS amplitude, set gain=1.0 / sqrt(2)

See also:

ramp()

7.2.7 mla.hardware

class mlaapi.hardware.Hardware(*parent*, *settings*, *hconfig*, *ip_address*, *firmware_path*, *hardware_version*, *eh=None*)

This class contains low-level functions for hardware setup and direct communication with the MLA. It also contains some auxiliary function that does not fit into any of the more user-oriented classes lockin, osc, arb or feeedback.

apply_settings()

Dummy function for forward compatibility with later API versions

configure_clk_status_ld()

Set the STATUS_LD pin of the clock circuit to be high when both PLLs are locked.

configure_memory(*ad0*, *ad1*, *da0*, *da1*)

Configure how the MLA should partition the available sample memory of 768 Megabytes.

Parameters

- ad0 – integer (default=128*MEGABYTE), sample memory for DMA ad0
- ad1 – integer (default=128*MEGABYTE), sample memory for DMA ad1
- da0 – integer (default=128*MEGABYTE), sample memory for DMA da0
- da1 – integer (default=128*MEGABYTE), sample memory for DMA da1

Returns
None

The memory is partitioned with respect to the DMA-channels (not the physical channels or ports). Adequate space must be reserved for the scatter-gather descriptors. Each SGD needs 64 bytes. For lockin data, the default is to have 200000 SGD, while for the other types of we usuallyhave only a few SGD.

Examples

```
>>> M = 1<<20 # One Megabyte
>>> # Optimize for one channel AD measurements
>>> mla.hardware.configure_memory(M, M, M, M, 760*M, M, M, M)
>>> # Set back the defaults
>>> mla.hardware.configure_memory(16*M, 16*M, 16*M, 16*M, 128*M, 128*M, 128*M,
↪128*M)
```

echo(*value='a'*)

Send a string to the MLA and wait for the MLA to return the same string.

Parameters
value – string

Returns
string

This function is useful to determine of the connection to the MLA is valid.

Examples

```
>>> mla.hardware.echo('Hello')
'Hello'
```

get_firmware_version()

Return the version of the firmware in the MLA hardware unit.

Returns
firmware version

Return type
integer

This functions queries the firmware itself (it is not reading the firmware filename).

Examples

```
>>> mla.hardware.get_firmware_version()
49
```

get_hardware_revision()

Return the hardware revision number of the unit

Returns
firmware version

Return type
integer

get_input_relay(*port*)

Obtain state of input relays

Returns
(se, nodamp, term, dc, gain, bypass, gain2)

Return type
tuple of booleans

get_nr_freqs_in()

Return the number of input frequencies that the MLA can analyze simultaneously.

Returns
number of frequencies

Return type
integer

This functions queries the firmware inside the MLA hardware.

Examples

```
>>> mla.hardware.get_nr_freqs_in()
42
```

See also:

get_nr_freqs_out(), get_firmware_version()

get_nr_freqs_out()

Return the number of out frequencies that the MLA can generate simultaneously.

Returns
number of frequencies

Return type
integer

This functions queries the firmware inside the MLA hardware.

Examples

```
>>> mla.hardware.get_nr_freqs_out()
42
```

See also:

get_nr_freqs_in(), get_firmware_version()

get_nr_input_ports()

Return the number of (initialized) physical input ports

get_output_relay(*port*)

Obtain state of output relay

Returns
boolean

get_overflow()

Read the overflow flags from the MLA and return a list of strings describing which flags are set.

Note: An overflow flag is active until it has be actively reset by hardware.Hardware.reset_overflow().

See also:

reset_overflow()

get_serial_number()

Return the serial number of the unit

Returns
serial number

Return type
str

get_temperature()

Read the temperature for the FPGA chip inside the MLA.

Parameters
n_samples – integer, the number of temperature measurements to make

Returns
temperatures in degrees Celsius

Return type
np.array(dtype=float)

Any temperature below 70 degrees should be considered safe. The chip is rated for 85 degrees but such high temperature for long periods may degrade the lifetime of the chip.

Examples

```
>>> mla.hardware.get_temperature(3)
array([ 42.09583664,  41.71133423,  42.14197693])
```

has_relays()

Test whether the current hardware has input and output relays.

Returns
(bool)

reset_overflow(*flags=4294967295*)

Reset overflow flags.

Parameters
flags – integer mask describing which flags should be reset,

Note: An overflow flag is active until it has be actively reset.

See also:

get_overflow()

set_LED(*val*)

Set which LEDs on the front panel should be turned on.

Parameters
val – integer, 0-15

Returns
None

The LEDs are assigned to numbers as follows: NETWORK=1, MODE=2, OVERLOAD=4, ERROR=8. The val parameter should be the sum of the LED that should be turned on.

Examples

```
>>> mla.hardware.set_LED(0xf) # Turn on all LED
>>> mla.hardware.set_LED(6)  # Turn on MODE and OVERLOAD, turn off the rest
>>> mla.hardware.set_LED(1)  # Turn on NETWORK, turn off the rest
```

set_ad_clock_divisor(*divisor*)

Set the clock frequency of the AD-converter for port IN 1 and IN 2.

Parameters
divisor – integer, 10-1045, the clock frequency will become 2.5 GHz/divisor

Returns
None

This clock frequency will also determine the speed of the lockin calculations.

Note: For high clock frequencies, it may be necessary to set_ad_clock_phase().

Examples

```
>>> mla.hardware.set_ad_clock_divisor(10)
>>> mla.hardware.set_ad_clock_phase(15)
>>> mla.hardware.set_ad_clock_divisor(50)
>>> mla.hardware.set_ad_clock_phase(0)
```

See also:

set_dac_clock_divisor(), set_slowad_clock_divisor() set_ad_clock_phase()

set_ad_clock_phase(*value*)

Set the timing register on the AD converter for port IN 1 and IN 2.

Parameters
value – integer, 0-15

Returns
None

Setting this value may be necessary when setting a high clock frequency to the ad converter for port IN 1 and IN 2. Typcal values are

ad_clock_divisor	ad_clock_phase
10	15
20-50	0

Examples

```
>>> mla.hardware.set_ad_clock_divisor(10)
>>> mla.hardware.set_ad_clock_phase(15)
>>> mla.hardware.set_ad_clock_divisor(50)
>>> mla.hardware.set_ad_clock_phase(0)
```

See also:

hardware.Hardware.set_ad_clock_divisor()

References

Put references, web links, or links to other sections of the user manual here.

set_clkout_divisor(*divisor*)

Set the clock frequency of REF CLK OUT port.

Parameters
divisor – integer, 1-1045, the clock frequency will become 2.5 GHz/divisor

Returns
None

Examples

```
>>> mla.hardware.set_clkout_divisor(25) # Set 100 MHz
>>> mla.hardware.set_ad_clock_phase(250) # Set 10 MHz
```

See also:

set_clkref_external_10MHz(), set_clkref_internal() _set_clkin_parameters()

set_clkout_power(*bool_power_on*)

Turn on or off the clock output signal at port REF CLK OUT

Returns
None

Examples

```
>>> mla.hardware.set_clkout_power(False)
>>> mla.hardware.set_clkout_power(True)
```

See also:

set_clkout_divisor(), set_clkout_type()

set_clkout_type(*typ*)

Set the output type of the port REF CLK OUT

Parameters
typ – integer, 0-14, see table below.

Returns
None

value	Description
0 (0x00)	Power down
1 (0x01)	LVDS
2 (0x02)	LVPECL (700 mVpp)
3 (0x03)	LVPECL (1200 mVpp)
4 (0x04)	LVPECL (1600 mVpp)
5 (0x05)	LVPECL (2000 mVpp)
6 (0x06)	LVCMOS (Norm/Inv)
7 (0x07)	LVCMOS (Inv/Norm)
8 (0x08)	LVCMOS (Norm/Norm)
9 (0x09)	LVCMOS (Inv/Inv)
10 (0x0A)	LVCMOS (Low/Norm)
11 (0x0A)	LVCMOS (Low/Inv)
12 (0x0C)	LVCMOS (Norm/Low)
13 (0x0D)	LVCMOS (Inv/Low)
14 (0x0E)	LVCMOS (Low/Low)

Note: Only the positive part of the diffential pair is connected to REF CLK OUT. It is first shunted to ground by 120 Ohm, and the AC-coupled with 100 nF

Examples

```
>>> mla.hardware.set_clkout_power(7)
>>> mla.hardware.set_clkout_power(1)
```

See also:

set_clkout_divisor(), set_clkout_power()

set_clkref_external_10MHz()

Lock the master clock to a 10 MHz signal at port REF CLK IN. This will reload the FPGA, see Notes!

Returns
None

Examples

```
>>> mla.hardware.set_clkref_external_10MHz()
>>> mla.hardware.set_clkref_internal()
```

Notes

All current MLA settings will be lost, see mla.reset() and mla.hardware.reload_fpga().

See also:

set_clkref_internal()

set_clkref_external_80MHz()

Lock the master clock to a 80 MHz signal at port REF CLK IN. This will reload the FPGA, see Notes!

Returns
None

Examples

```
>>> mla.hardware.set_clkref_external_80MHz()
>>> mla.hardware.set_clkref_internal()
```

Notes

All current MLA settings will be lost, see mla.reset() and mla.hardware.reload_fpga().

See also:

set_clkref_internal()

set_clkref_internal()

Lock the master clock to the internal clock reference. This will reload the FPGA, see Notes!

Returns
None

Examples

```
>>> mla.hardware.set_clkref_external_10MHz()
>>> mla.hardware.set_clkref_internal()
```

Notes

All current MLA settings will be lost, see mla.reset() and mla.hardware.reload_fpga().

See also:

set_clkref_internal()

set_dac_auxdac(*port*, *value*, *unit='calibrated'*, *state='on'*)

Set a DC offset to port OUT 1 or OUT 2.

Parameters

- port – 1 or 2
- value – float, if unit='digital', value must be 0-1023
- unit – 'calibrated' or 'digital'
- state – 'on' or 'off', this option can be used to turn off the axudac

Returns
None

This function is connected to a separate DA-converter, which is connected to the common mode pin of the differential amplifier at the outout stage.

Examples

```
>>> mla.hardware.set_dac_auxdac(1, 0.7)
>>> mla.hardware.set_dac_auxdac(1, 0, unit='digital')
```

See also:

Calibration, set_slowdac()

set_dac_clock_divisor(*divisor*)

Set the clock frequency of the DA-converter for port OUT 1 and OUT 2.

Parameters
divisor – integer, 10-1045, the clock frequency will become 2.5 GHz/divisor

Returns
None

Examples

```
>>> mla.hardware.set_dac_clock_divisor(25) # Set 100 MHz
>>> mla.hardware.set_dac_clock_divisor(50) # Set 50 MHz
```

See also:

set_ad_clock_divisor(), set_slowad_clock_divisor()

set_dac_fsc(*port*, *value=512*)

Set the "full scale current" of the DA-converter. Use this command to configure the maximum output voltage on ports OUT 1 and OUT 2.

Parameters

- port – 1 or 2
- val – 0-1023 (0->8.66 mA, 512->20.0 mA (default), 1023->31.66 mA)

Returns
None

Warning: Setting another value than 512 will make the calibration invalid.

Examples

```
>>> mla.hardware.set_dac_fsc(1, 1023)
>>> mla.hardware.set_dac_fsc(1, 512)
```

set_dac_interpolation(*interpolation_type*)

Configure how the DAC should interpolate the data stream

Parameters
interpolation_type – integer, 0 = No interploation, 1 = linear, 2 = quadratic

Returns
None

Examples

```
>>> mla.hardware.set_dac_interpolation(2)
```

set_input_relay(*port*, *se=False*, *nodamp=False*, *term=False*, *dc=False*, *gain=False*, *bypass=False*, *gain2=False*, *event=True*)

Manually configure analog input stage. Switch states are displayed in the MLA GUI. See MLA manual for description of the switch states.

Parameters

- port – int, input port number.
- se – bool, signal type.
- nodamp – bool, series restance.
- term – bool, input termination.
- dc – bool, ac/dc coupling.
- gain – bool, 1st stage gain.
- bypass – bool, bypass first gain stage.
- gain2 – bool, 2nd stage gain.
- event – bool, make the eventhandler trigger the EVT_INPUT_RELAY_UPDATED event.

Returns
None

Note: Eeach call of this function sets all relays. Default relay state is False for each relay.

---

set_output_multiplexer(*channel_1*, *channel_2*)

Deprecated, please use set_output_type() instead. Control weather the output ports shall output from the lockin waveform generators, or the arbitrary waveform generator.

Parameters

- channel_1 – "off", "arb", or "lockin"
- channel_2 – "off", "arb", or "lockin"

set_output_relay(*port*, *bypass=False*, *event=True*)

Manually configure analog output stage.
Use to control the range of the output signal.

Parameters

- port – int, input port number.
- bypass – bool, bypass 2nd stage output amplifier
- event – bool, make the eventhandler trigger the EVT_OUTPUT_RELAY_UPDATED event.

Returns
None

set_output_type(*value*, *port*)

Control weather the output ports shall output from the lockin waveform generators, or the arbitrary waveform generator.

Parameters

- value – "off", "arb", "lockin", "lockin+arb" or None
- port – 1 or 2

set_slowad_clock_divisor(*divisor*)

Set the clock frequency of the AD-converter for port IN 3 and IN 4.

Parameters

divisor – integer, 40-1045, the clock frequency will become 2.5 GHz/divisor

Returns

None

Examples

```
>>> mla.hardware.set_slowad_clock_divisor(100)
>>> mla.hardware.set_ad_clock_phase(50)
```

See also:

set_dac_clock_divisor(), set_ad_clock_divisor()

set_slowdac(*port*, *value*, *unit='calibrated'*)

Apply a voltage at port OUT A, OUT B or OUT C (or the internal out D).

Parameters

- port – 'A', 'B', 'C', 'D' or integer 1-4
- value – float, The output value in the specified unit (see unit argument)
- unit – 'calibrated' or 'digital', if unit='digital', value should be -32768 to 32767

Returns

None

Note: In the standard AFM configuration, port A is used for feedback and cannot be set by this method. To deactivate the feedback and activate port 'A' for set_slowdac, run mla.feedback.deactivate_slowdac().

Examples

```
>>> mla.hardware.set_slowdac('B', 1.7)
>>> mla.hardware.set_slowdac('B', -3.5)
>>> mla.hardware.set_slowdac('B', 0)
```

See also:

set_slowdac_toggle(), *Calibration*

set_slowdac_toggle(*val1*, *val2*, *unit='calibrated'*, *enable=1*)

Setup port OUT B to toggle between val1 and val2 each time a trigger event is detected on port TRIG IN 1.

Parameters

- val1 – float (see unit argument)
- val2 – float (see unit argument)
- unit – 'calibrated' or 'digital', if unit='digital', value should be -32768 to 32767

- enable – 0 or 1, to disable or enable this functionality

Returns
None

At the first trigger event after this function is executed, the value of port B will be set to val1. At the second trigger event val2, etc.

Examples

```
>>> # Toggle between 1.5 V and 3.3 V
>>> #mla.hardware.set_slowdac_toggle(1.5, 3.3)
>>> # Toggle between 0 and full scale output
>>> #mla.hardware.set_slowdac_toggle(0, 32767, unit='digital', enable=1)
>>> # Disable toggling
>>> #mla.hardware.set_slowdac_toggle(0, 0, unit='digital', enable=0)
```

See also:

set_slowdac(), *Calibration*

References

Put references, web links, or links to other sections of the user manual here.

upload_calibration(*filepath*, *target_filename=None*)

Upload a calibration file to the MLA.

Parameters

- filepath – String, the full path to the file that should be uploaded.
- target_filename – String or None, set a different name of the uploaded file. If None, use the same name as the source.

Examples

```
>>> mla.hardware.upload_calibration('c:\calibration.ini')
>>> mla.hardware.upload_calibration('c:\calibration.ini', 'cal.ini')
```

See also:

mlaapi.unit_converter.UnitConverter.set_calfile()

wait_for_idle()

Wait for MLA to process all previously sent commands and reach idle state.

7.3 MLA GUI

The Graphical User Interface (GUI) for the MLA has a *Python shell panel* and a *Script panel* from which you can execute all commands in the *MLA API*. But it is also possible to control the MLA GUI itself from a Python script. For example, you can program the GUI to:

- Set which panels should be visible, and their size and position
- Control GUI objects such as text fields and check boxes
- Configure the plots

The MLA GUI is controlled through it's hierarchy of objects. At the top level there is the main controller, called *main*. There is also *mla_globals*, which is a container for global variables, and the *mla* object itself. Only a selected subset of the full MLA GUI programming interface is documented under the links below. If you need to access a function that is not documented here, please contact Intermodulation Products.

7.3.1 mla_gui

class mlaapi.mla_gui.ErrPanel(*parent*)

Class for having a panel around the error status field. This way we can set the background color and flash it.

class mlaapi.mla_gui.IMPStatusBar(*parent*)

class mlaapi.mla_gui.ImGUI(*parent*, *settings*, *id=-1*, *title=''*, *pos=wx.Point(-1, -1)*, *size=wx.Size(-1, -1)*, *style=675290688*)

This class contains the menu bar and handles the panels.

hide_all_panels(*evt=None*)

Note: The panels are not closed, only hidden. When a panel is shown again, it will look the same as before it was hidden.

Examples

```
>>> main.gui.hide_all_panels()
```

See also:

show_panel()

set_docksize(*panel*, *size*)

A dock can contain one or more docked panels. This function sets the size of the dock that contains a specified panel.

Parameters

- panel – panel
- size – integer size of the dock in pixels

Examples

```
>>> main.gui.set_docksize(main.message_log, 300)
>>> main.gui.set_docksize(main.shell, main.gui.GetSize()[0]/2) #Use half the horizontal␣
↪window size
```

show_panel(*panel*, *pos=None*)

Parameters

- panel – panel to show
- pos – string or list of strings that specify the position. The string should represent a member function call of a AuiPaneInfo object. See the examples and the link under References.

Examples

```
>>> main.gui.show_panel(main.scopepanel)
>>> main.gui.show_panel(main.shell)
>>> main.gui.show_panel(main.scriptpanel)
>>> main.gui.show_panel(main.shell, pos='Left()')
>>> main.gui.show_panel(main.message_log, pos='Bottom().Dock()')
>>> main.gui.show_panel(main.lockinpanel)
>>> main.gui.show_panel(main.lockin_setup_panel)
>>> main.gui.show_panel(main.frequency_count_panel)
>>> main.gui.show_panel(main.fsweeppanel)
>>> main.gui.show_panel(main.message_log, pos=('Dock()', 'Bottom()', 'Layer(1)'))
```

See also:

hide_all_panels()

References

http://wxpython.org/Phoenix/docs/html/lib.agw.aui.framemanager.AuiPaneInfo.html

class mlaapi.mla_gui.MyAuiManager(**args*, ***keys*)

ClosePane(*pane_info*)

Destroys or hides the pane depending on its flags.

Parameters
pane_info – a AuiPaneInfo instance.

Override because ClosePane will dock a floating pane for some reason, and we don't want that. So we let ClosePane do its thing, and then we set the pane as floating again. Possibly there's a better way... //Riccardo 2018-04-11

7.3.2 Script panel

class mlaapi.scriptpanel.ScriptPanel(*parent*, *mla*, *shell*, *scriptplot*, *id=-1*, *stdout=None*, *stderr=None*)

The ScriptPanel contains a text editor and a toolbar to open, save and execute scripts.

open_file(*filename*)

Open a file in the script editor. The file will not be executed.

Parameters
filename – Full path, or path relative to the system 'mla_scripts' folder.

Note: When a script runs, the path to the current script is stored in the variable __scriptfile__.

Note: If a file is already opened it will be closed without saving.

Examples

```
>>> main.scriptpanel.open_file('my_script.py')
>>> main.scriptpanel.open_file('built-in    emperature_test.py')
>>> main.scriptpanel.open_file('c:\myscripts\myscript.py')
>>> main.scriptpanel.open_file(__scriptfile__)
```

7.3.3 Scripting utilities

This module collects various utility functions that can be useful in the script editor in the MLA software.

mlaapi.scriptutils.generate_filename(*foldername_format='%Y-%m-%d'*, *filename_format='%H_%M_%S'*)

Generate the complete path to a new file in the user data folder containing the data as folder name and time as filename. New subfolders are created if necessary.

Parameters

- foldername_format – string, will be passed to time.strftime() to generate the folder name
- filename_format – string, will be passed to time.strftime() to generate the file name

Returns
The full path to a new file.

Return type
string

Examples

```
>>> scriptutils.generate_filename() + '.png'
'C:\IMP Sessions and Settings\sessions\2015-09-04\08_45_30.png'
>>> scriptutils.generate_filename(foldername_format='sample_1', filename_format=rig2_
↪%Y-%m-%d__%H_%M_%S) + '.txt'
'C:\IMP Sessions and Settings\sessions\sample_1\rig2_2015-09-04__08_56_37.txt'
```

7.3.4 Shell panel

class mlaapi.impgui.ShellPanel(*parent*, *id=-1*)

Panel that contains a python shell.

add_text(*txt*)

Adds a text to the prompt of the python shell. The text command is not executed.

Parameters
txt – string

Examples

```
>>> main.shell.add_text('mla.run_calibration_sequence()')
```

mlaapi.impgui.flash_color(*widget*, *color='yes'*, *time=0.8*, *reset_color='#FFFFFF'*)

Change background color of a wx widget for a certain amount of time

Parameters

- widget (a wx window object (e.g. ex.TextCtrl)) –
- color (string) – 'yes' (default) for a green color 'no' for red or any argument accepted by widget.SetBackgroundColour
- time (float) – Time in seconds before resetting color
- reset_color (color) – Color which will be set after time (default: "#FFFFFFF")

7.3.5 Lockin setup panel

7.3.6 Oscilloscope panel

class mlaapi.scopepanel.Marker(*channel: int*, *x: float*, *value: float = nan*, *marker_style: matplotlib.markers.MarkerStyle = <matplotlib.markers.MarkerStyle object at 0x00000000177FBD00>*, *line: Union[matplotlib.lines.Line2D, NoneType] = None*, *unit_x: str = ''*, *unit_value: str = ''*)

class mlaapi.scopepanel.PlotPanel(*parent*, *eh*, *name*, *position*, *layer*, *n_channels*, *n_lockin_ports*, *id=-1*)

Class for the Oscilloscope plot panel with time and spectrim plots.

set_limits(*x1_min*, *x1_max*, *y1_min*, *y1_max*, *x2_min*, *x2_max*, *y2_min*, *y2_max*)

Set the axes limits of the time and freqeuncy plots.

Parameters

- x1_min – float [ms]
- x1_max – float [ms]
- y1_min – float [V]
- y1_max – float [V]
- x2_min – float [kHz]
- x2_max – float [kHz]
- y2_min – float [dBV]
- y2_max – float [dBV]

Examples

```
>>> main.scopepanel.set_limits(0, 0.1, -1, 1, 0, 200, -140, 10)
```

7.3.7 Lockin plot panel

class mlaapi.lockin_plotpanel.Frequency_counting_panel(*parent*, *eh*, *name*, *position*, *layer*, *n_channels*, *id=-1*)

set_subplot_layout(*nr*, *nc*)

Set how many rows and columns of subplots the panel should have.

Parameters

- nrows – integer number of rows
- ncols – integer number of columns

Examples

```
>>> main.lockinpanel.set_subplot_layout(nrows=2, ncols=2)
```

class mlaapi.lockin_plotpanel.LockinPlotPanel(*parent*, *eh*, *name*, *position*, *layer*, *n_channels*, *id=-1*)

class mlaapi.lockin_plotpanel.PlotPanel(*parent*, *eh*, *name*, *position*, *layer*, *n_channels*, *id=-1*)

Class for handling lockin plots

set_axis_properties(*row*, *col*, *tone_indices=None*, *quantity='Amplitude'*, *yscale_type='Linear'*, *ymin=None*, *ymax=None*, *legend_pos=-1*, *history_length=30*, *title=''*)

Set the properies of one of the subplots in the plot panel.

Parameters

- row – integer, identifies which subplot to configure
- col – integer, identifies which subplot to configure
- tone_indices – list of integers, set which tones to plot
- quantity – “Amplitude”, “Phase”, “I” or “Q”, see table below
- yscale_type – “Linear”, Log” or “Wrap”, see table below
- ymin – float, minimum value of the y-axis
- ymax – float, maximum value of the y-axis
- legend_pos – integer -1-10, position of the legend, see table below
- history_length – float [seconds], how long history to plot (i.e. the scale of the x-axis)

quantity	Description
Amplitude	amplitude
Phase	phase
I	In-phase component
Q	Quadrature-phase component

yscale_type	Description
Linear	Linear
Log	Logarithmic
Wrap	Wrap at limits

legend_pos	Description
-1	No legend
0	best
1	upper right
2	upper left
3	lower left
4	lower right
5	right
6	center left
7	center right
8	lower center
9	upper center
10	center

Examples

```
>>> main.lockinpanel.set_axis_properties(row=0, col=0,
        tone_indices = [1, 3, 7],
        quantity = "Amplitude",
        yscale_type = "Log",
        ymin = 1e-6,
        ymax = 10,
        legend_pos = 1)
```

set_subplot_layout(*nrows=None*, *ncols=None*)

Set how many rows and columns of subplots the panel should have.

Parameters

- nrows – integer number of rows
- ncols – integer number of columns

Examples

```
>>> main.lockinpanel.set_subplot_layout(nrows=2, ncols=2)
```

class mlaapi.lockin_plotpanel.PlotPanelBase(*parent*, *eh*, *name*, *position*, *layer*, *id=-1*)

request_draw()

Queue new draw only if previous request is done. Safe to call from non-main thread.

class mlaapi.lockin_plotpanel.UserPlotPanel(*parent*, *eh*, *name*, *position*, *layer*, *id=-1*)

7.3.8 Line colors

The colors in the "Lockin history" plot are selected from a predefined list to ensure maximal color contrast between each line. However, depending on the physical or logical topology of the measurement setup other color schemes may be desired. To edit the colors, set the mla_globals.lockin_colors variable from the shell or a script. Below are a few examples:

```
# Cycle the colors red, green, blue, cyan, magenta, yellow, white
mla_globals.lockin_colors = 'rgbcmyw'
```

(continues on next page)

(continued from previous page)

```
# Cycle the colors red, green, blue
mla_globals.lockin_colors = ('#ff0000','#00ff00', '#0000ff')

# Cycle the colors cyan, magenta, yellow
mla_globals.lockin_colors = ((0, 1, 1), (1, 0, 1), (1, 1, 0))

# Get a unique color for each frequency from the jet colormap. See all available colormaps here:
# http://matplotlib.org/examples/color/colormaps_reference.html
mla_globals.lockin_colors = matplotlib.cm.jet(np.linspace(0,1,mla.lockin.nr_input_freq))
```

7.4 example scripts

Here follows some examples scripts which can be run from the *script panel* in the MLA GUI. They serve as starting points for designing your own measurement scripts. These examples are available in the **settings/mla_scripts/built-in** folder in the *user folder*.

In every script the MLA API is instantiated as the Python object called `mla` and all commands are accessed in the code with the form `mla.comand`. All commands used in these scripts are documented in detail in *MLA API*

7.4.1 setup the GUI

Panels can automatically be displayed or hidden using scripts. Most panels are accessed through the *main.* object. A script can obtain its own filename with *__scriptfile__*.

```
# Built-in script. Use the "Save as" button and save under a different filename before editing.


# Set which panels should be visible
main.gui.show_panel(main.scriptpanel)
main.gui.show_panel(main.scriptplot)
main.gui.show_panel(main.shell)
main.gui.show_panel(main.message_log, pos=('Dock()', 'Bottom()', 'Layer(1)'))
main.scriptpanel.open_file(__scriptfile__)
```

7.4.2 plotting in the GUI

Plotting uses matplotlib and the script plot panel. A pre-existing Figure object is first obtained from *scriptplot.fig* after which regular matplotlib syntax is used. To draw the figure on the screen the function *scriptplot.draw* has to be called. As wxPython only allows the main thread to perform to access the GUI this command has to be queued in the main loop using *wx.CallAfter*. Failure to do so can lead to unstable behavior and crashing of the software.

```
# Built-in script. Use the "Save as" button and save under a different filename before editing.

# Perform plotting in the Script Plot panel and save the
# resulting figure under an automatically generated filename


fig = scriptplot.fig
fig.clear()

ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel('x')
ax.set_ylabel('t')
```

(continues on next page)

(continued from previous page)

```
ax.set_title('Figure 1')
t = np.linspace(0, 1, 1000)
y = np.sin(2 * np.pi * t)
line = ax.plot(t, y, 'r')[0]

fig.tight_layout()
filename = scriptutils.generate_filename() + '.png'
wx.CallAfter(scriptplot.draw)
wx.CallAfter(fig.savefig, filename)
```

7.4.3 configuring the lockin

This script demonstrates configuring the lockin using explicit lists. Python lists as well as numpy arrays can be used as an argument when setting properties such as frequency, amplitude and phase. If the length of the argument is less than the number of available tones (i.e. `mla.lockin.nr_output_freq()`) that property will be set to zero for the remaining, unspecified, tones.

```
# Built-in script. Use the "Save as" button and save under a different filename before editing.


# Configure lockin
freqs = [10000, 10100, 10200]  # Hz
df = 100  # Hz
phases = [0, 90, 180]  # degree
ampls = [0.01, 0.02, 0.03]  # V
out_1_mask = [1, 1, 1]
out_2_mask = 0

# Tune frequencies
freqs_tuned, df_tuned = mla.lockin.tune1(freqs, df)

# Program MLA
mla.reset()
mla.lockin.set_output_mask(out_1_mask, port=1)
mla.lockin.set_output_mask(out_2_mask, port=2)
mla.lockin.set_phases(phases, 'degree')
mla.lockin.set_amplitudes(ampls)
mla.lockin.set_frequencies(freqs_tuned)
mla.lockin.set_df(df_tuned)
```

7.4.4 lockin measurement loop

This example takes a lockin measurement and displays the amplitudes in a continuously updating figure. The measurement is running an infinite loop which is broken by changing the Boolean *scriptpanel.is_running*. The value of this variable will change when the user clicks the Run/Stop button in the Script panel.

The Boolean *scriptplot.is_drawing* is automatically set to False when *scriptplot.draw* finishes. If a previous plot was not finished, plotting is skipped to ensure that the GUI remains responsive and the computer is not overloaded with plot requests. To speed up plotting the command *ax.plot* should be avoided in the loop as this command redraws all axes and ticks etc. Rather, update only the data in an existing matplotlib *line*.

```
# Built-in script. Use the "Save as" button and save under a different filename before editing.

```

(continues on next page)

(continued from previous page)

```
freqs = mla.lockin.get_frequencies()

scriptplot.is_drawing = False
fig = scriptplot.fig
fig.clear()
ax = fig.add_subplot(1, 1, 1)
line = ax.plot(freqs * 1e-3, np.zeros_like(freqs), 'x')[0]
ax.set_xlabel('Frequency (kHz)')
ax.set_ylabel('Amplitude ({})'.format(mla.convert.get_AD_unit()))

mla.lockin.start_lockin()
while True:
    mla.lockin.wait_for_new_pixels(1)
    pixels, meta = mla.lockin.get_pixels(1)
    line.set_ydata(np.abs(pixels))
    scriptplot.request_draw()
    if not scriptpanel.is_running:
        break
mla.lockin.stop_lockin()
```

7.4.5 frequency sweep

He we demonstrate changing a parameter in a measurement loop. The lockin frequency is stepped through an array *f_hz* of specified values. After tuning, each value of the bandwidth is set using the command *mla.lockin.set_df(df, wait_for_effect=False)*, followed by the command *mla.lockin.set_frequencies(f, wait_for_effect=True)*. Python does not have to wait for the MLA to reply before it goes on to the second configuration command. However, Python should wait after the last configuration command (i.e. *wait_for_effect=True*) before it goes on to get new data from the MLA. This speeds up configuration, while ensuring that the configuration is actually in effect when the measurement data is collected.

Note that the lockin is started with the command *mla.lockin.start_lockin(cluster_size=1)*, specifying that only one lockin data packet is sent with each Ethernet package. This speeds up the loop by reducing the latency between the end of measurement and transfer to the computer. The feature is useful when we are interested in fast update after only one lockin measurement. However, a small cluster size will reduce the total bandwidth when a data stream of many lockin measurements is desired.

This example demonstrates stepping any lockin parameters. A frequency sweep is most easily done in the *MLA GUI* using the *frequency sweep panel*. In your own scripts, use the function lockin.Lockin.frequency_sweep(), which is optimized for speed, and has the option for multifrequency sweeping is available.

```
# Simple frequency sweep using Python loop to step through parameters and call MLA␣
↪commands

# User parameters
f_khz = np.linspace(1, 10000, 1001)
df = 1000.
drive_amp = 0.1

# Setup MLA
mla.lockin.set_df(df)
mla.lockin.set_amplitudes(drive_amp)
mla.lockin.set_output_mask(1, port=1)

# Allocate memory for result
amps = np.zeros_like(f_khz)
amps[:] = np.nan
```

(continues on next page)

(continued from previous page)

```
phases = np.zeros_like(f_khz)
phases[:] = np.nan

# Setup GUI
main.gui.show_panel(main.scriptplot)

# Initialize plots
scriptplot.fig.clear()  # remove if you want multiple sweeps in one plot
ax, axp = scriptplot.fig.subplots(2,1, sharex=True)

ax.set_xlabel('Frequency [kHz]')
ax.set_ylabel('Amplitude [V]')
ax.set_title('Frequency sweep')
ax.set_xlim([f_khz.min(), f_khz.max()])
line = ax.plot(f_khz, amps)[0]

axp.set_xlabel('Frequency [kHz]')
axp.set_ylabel('Phase [rad]')
axp.set_xlim([f_khz.min(), f_khz.max()])
linep = axp.plot(f_khz, phases)[0]

# Start lockin should have cluster_size=1
# since we do not want many packages buffered
# on MLA before they are transmitted
mla.lockin.start_lockin(cluster_size=1)
t0 = time.time()
for ii, f in enumerate(f_khz):
    # Set parameters in loop
    f_tuned, df_tuned = mla.lockin.tune1(f * 1e3, df)
    f_khz[ii] = f_tuned * 1e-3
    mla.lockin.set_df(df_tuned, wait_for_effect=False)
    mla.lockin.set_frequencies(f_tuned, idx=0, wait_for_effect=True)  # wait for effect␣
↪on last MLA setting

    # Recieve new lockin data
    mla.lockin.wait_for_new_pixels(1)
    pix, meta = mla.lockin.get_pixels(1)
    amps[ii] = np.abs(pix[0])
    phases[ii] = np.angle(pix[0])

    # Update plots
    line.set_data(f_khz[:ii + 1], amps[:ii + 1])
    linep.set_data(f_khz[:ii + 1], phases[:ii + 1])
    scriptplot.autoscale_y(ax)
    scriptplot.autoscale_y(axp)
    scriptplot.request_draw()
    if not scriptpanel.is_running:
        break

t1 = time.time() - t0
print('Run time: ' + str(t1))
mla.lockin.stop_lockin()

# Final update of plots
line.set_data(f_khz, amps)
linep.set_data(f_khz, phases)
```

(continues on next page)

(continued from previous page)

```
scriptplot.autoscale_y(ax)
scriptplot.autoscale_y(axp)
wx.CallAfter(scriptplot.draw)
filename = scriptutils.generate_filename() + '.png'
wx.CallAfter(scriptplot.fig.savefig, filename)
```

7.5 stand-alone scripts

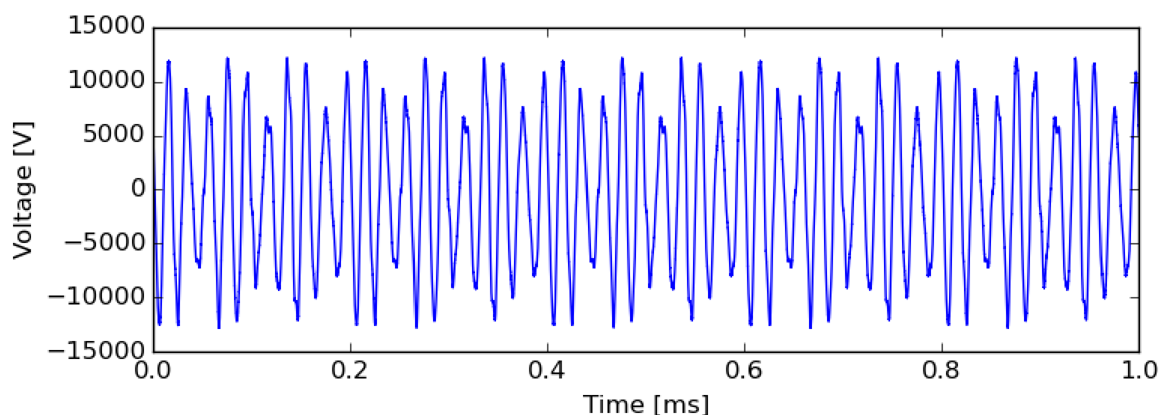
Scripts do not have to be run from the *script panel*. A stand-alone python script is demonstrated in the example below. The script configures the lockin to make a frequency comb, outputs the comb, measures the comb as a time stream, and plots the measured data in both the time and frequency domain in a figure which is displayed below. Connect **OUT 1+** to **IN 1+** on the MLA before running this script.

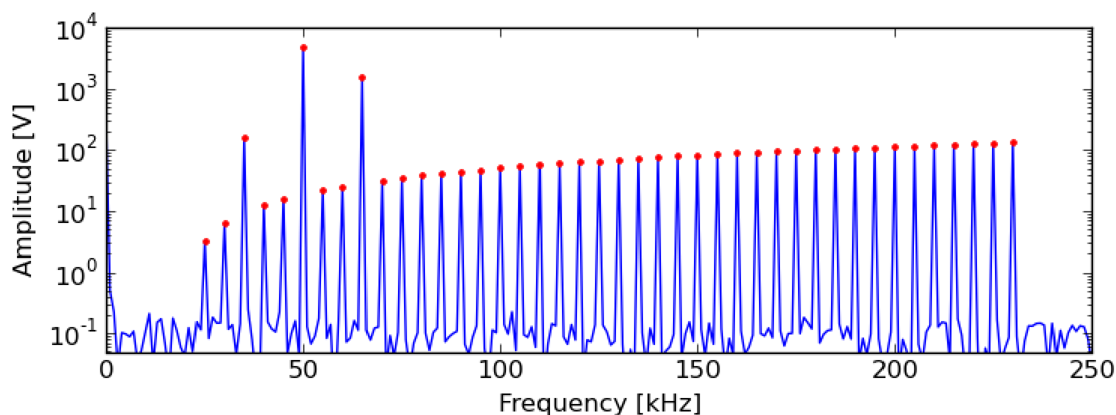
```
# This script demonstrates some basic features of the MLA API together
# with numpy and matplotlib. The code can be executed line-by-line in a
# python or ipython console, or run as a script with
# "python mla_api_demo.py".
# It will set up the MLA to produce a number of tones and measure
# their response, both in time- and frequency mode. The time mode
# measurement is Fourier transformed and compared with the time mode
# measurement. The result is visualized as a matplotlib plot that should
# appear on screen.

import sys

sys.path.append("PATH TO FOLDER CONTAINING mla_api HERE")

if __name__ == '__main__':  # This line is necessary on Windows operating
    # systems because the MLA is using multiple processes.
    import matplotlib.pyplot as plt
    import numpy as np

    from mlaapi import mla_globals
    from mlaapi import mla_api

    settings = mla_globals.read_config()
    mla = mla_api.MLA(settings)
    mla.connect()
    mla.lockin.start_lockin()
    mla.lockin.set_df(1000)
    mla.lockin.set_frequencies_khz(25 + np.arange(mla.lockin.nr_output_freq) * 5)
    mla.lockin.set_amplitudes(np.arange(1, mla.lockin.nr_output_freq + 1) * 0.0002)
    mla.lockin.set_amplitudes([0.01, 0.3, 0.1], idx=[2, 5, 8])
    mla.lockin.set_output_mask(np.ones(mla.lockin.nr_output_freq))
    mla.osc.start_time_stream(mla.lockin.samples_per_pixel)
    y = mla.osc.get_data(mla.lockin.samples_per_pixel)
    t = np.arange(len(y)) / mla.osc.get_samplerate() * 1e3
    (fig, (ax1, ax2)) = plt.subplots(2, 1)
    ax1.plot(t, y)
    ax1.set_xlabel('Time [ms]')
    ax1.set_ylabel('Voltage [V]')
    y_fft = np.fft.rfft(y) / len(y)
    f_fft_khz = np.fft.rfftfreq(len(y), d=1. / mla.osc.get_samplerate()) * 1e-3
    p = mla.lockin.get_last_pixel()
```

(continues on next page)

(continued from previous page)

```
    a = np.sqrt(p[::2]**2, p[1::2]**2)
    f = mla.lockin.get_frequencies_khz()
    ax2.semilogy(f_fft_khz, abs(y_fft), f, a, '.r')
    ax2.set_xlim([0, 250])
    ax2.set_ylim([5e-2, 1e4])
    ax2.set_xlabel('Frequency [kHz]')
    ax2.set_ylabel('Amplitude [V]')
    fig.tight_layout()
    mla.lockin.stop_lockin()
    mla.disconnect()
    plt.draw()
    plt.show()
```





CHAPTER

EIGHT

FILES FOLDERS AND CONFIGURATIONS

8.1 folders

The MLA software is working in two distinct folders. The locations of these two folders are selected during the installation.

8.1.1 program folder

The program folder typically has the location **C:\Program Files\IMP MLA**. This folder contains static files that never change under normal operation, only when the software is updated. For example the executable program files, the firmware files, icons, built-in scripts, etc. This folder is usually marked as "read only" and any change would require administrator privileges. The path to the program folder is selected in the software installation wizard. If it is moved after the installation the short-cuts in the start menu will point incorrectly and would have to be fixed manually. This folder does not need to be backed up, since it can be recreated by re-running the MLA installation program.

8.1.2 user folder

The user folder is typically has the location **C:\IMP Data and Settings**. This folder contains all files that are changed by the user during normal operation, most notably measurement data files, but also configuration files and scripts. The path to the user folder is selected in the software installation wizard. If the user folder is moved or renamed, the first line of **paths.ini** in the *program folder* must be updated accordingly (which usually requires administrator privileges). Users are encouraged to keep a backup of this folder. When the software is updated, this folder is left untouched.

8.2 configuration files

The MLA and the MLA software are configured by a number of different files with different roles. They are processed in the following order:

1. **paths.ini** tells the location of **mla_config.ini**.
2. **mla_config.ini** will then tell which firmware to load and which calibration to use.
3. When the software if fully loaded, a start-up script that is specified in **mla_config.ini** will be executed.
4. The startup script can then call other scripts if necessary. None of the default startup scripts are calling other scripts.

Below are more detailed information about these files.

8.2.1 paths.ini

paths.ini should exist in the same folder as the main executable file (**imp-mla.exe** or **imp-mla.py**). It contains two lines. The first line is the full path to the user folder and the second line is the full path to the user manual.

8.2.2 mla_config.ini

mla_config.ini should be located in the *user folder*. It contains the basic settings for the MLA and MLA software using the well-established INI file format (see for example http://wikipedia.org/wiki/INI_file). **mla_config.ini** specifies:

- Which firmware file to load in to the FPGA.
- Which calibration file to use.
- Which startup script to run.

As long as a configuration item is kept at its default value, it will not be written to **mla_config_spec.ini** where the full specification of all available parameters, their default values, and valid values are stored. **mla_configspec.ini** is also stored in the *program folder*. Under normal operation the user should not have to edit **mla_config.ini** manually. If you do edit this file, the MLA software should be shut down before editing, to avoid undesired over-writing when exiting the software.

8.2.3 firmware file

A firmware file is used to program the FPGA inside the MLA hardware. This is where the powerful real-time digital signal processing occurs, enabling the multifrequency lock-in capabilities. Different firmware files can be optimized for different measurement applications, for example configuring the number of tones in a multifrequency lock-in measurements. The firmware files are found in the **firmware** sub-folder of the *program folder*.

A firmware filename can have the format: **FPGA_NNN_AA_BB_YYYYMM.top.bit**, in which case the codes have the following meaning:

- **NNN**, version number of the code, incremented by 1 every time the code base for the firmware is changed.
- **AA**, number of input tones that can be analyzed.
- **BB**, number of output tones that can be generated.
- **YYYYMM**, hardware revision number that is compatible with this firmware.

8.2.4 calibration files

The calibration files are described in the section *Calibration*.

8.2.5 start-up script

At startup the *MLA GUI* can run a script specified by the key **startup_script** in the file **mla_config.ini**. To change the start-up script you can edit the value of this key, or change it from the GUI by opening the desired script in the *script panel* and pressing the `startup` icon in the toolbar.

CHAPTER

NINE

ANALOG INTERFACES

This chapter gives more detailed information about the analog interfaces of the 3rd generation MLA™.

The two signal output ports and the four signal input ports are controlled in the *MLA GUI* via the *analog panel*.

9.1 signal outputs

OUT 1 and **OUT 2** are two high speed outputs with 500 MSam/sec and 16 bit resolution. These ports are are always differential and biploar. The output voltage will appear between the + port and ground (SMA shield). The same same voltage with the opposite sign appears between the - port and ground. Therefore, if you use these ports differentially, the voltage difference between and + and - ports will be twice the set value. These ports have 50 Ohm output impedance and they have two ranges:

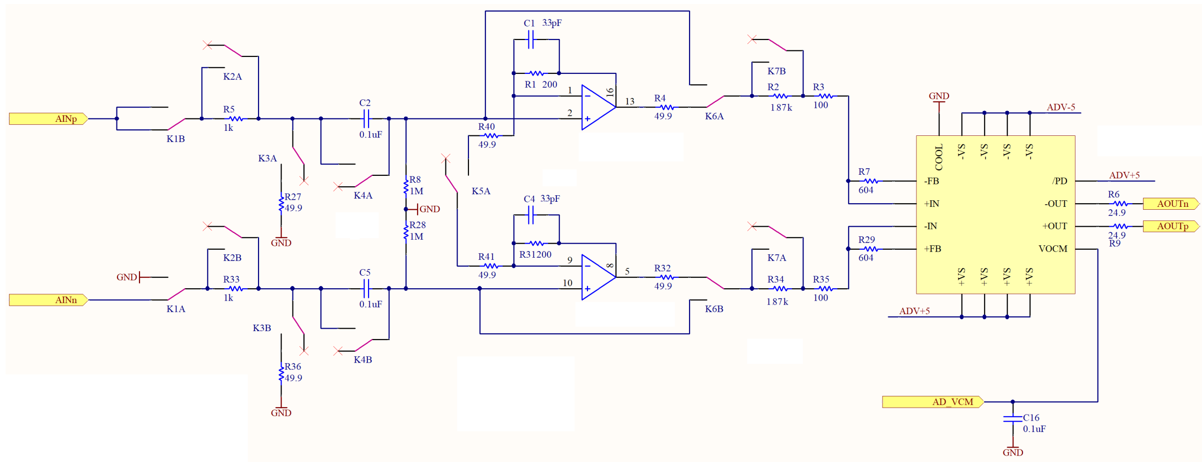
> $\pm$ 2 V range, maximum current 50 mA.
>
> $\pm$ 12 V range, maximum current 350 mA.

> **Caution:** The 12 V range is rather powerful and you can burn stuff with it. Be careful with what you are driving when using the 12 V range on these ports.

9.2 signal inputs

IN 1 and **IN 2** are high-speed (250 MSam/sec) with 14 bit resolution. **IN 3** and **IN 4** have moderate speed (50 MSam/sec) with 16 bit resolution. All ports can measure either differential or single-ended signals. When measuring single-ended the - input sees an open circuit (but the - input amplifier is grounded internally). All ports have switchable range settings, AC or DC coupling, 50 ohm or 1M ohm input impedance. Making the optimal measurement involves managing trade-offs that effect the noise, gain, bandwidth and intermodulation distortion. You should test different configurations of the MLA™-3 analog interface to see what works best for your measurement.

You can configure the signal inputs from the *MLA GUI* via the *analog panel*. When using the *analog panel* in the `Configuration` mode, the check boxes under `Manual switch control` display the state of the input relays. To control the input analog interface in your own program via the *MLA API*, use the function `hardware.Hardware.set_input_relay()`, setting every input argument (switch name) to its the desired state, as described in the table below.



Switch Description	Schematic	API name	state=True	state=False
Signal Type	K1	se	single ended	differential
Series Resistance[1]	K2	nodamp	no series R	1K series R
Input termination	K3	term	50 ohm	1M ohm
AC/DC Coupling	K4	dc	DC coupling	AC coupling
1st stage gain	K5	gain	x 1	x 5
Bypass	K6	bypass	bypass 1st stage	use 1st stage
2nd stage gain	K7	gain2	x 5	x 0.5

9.3 aux outputs

OUT A to **OUT D** are four slower ports are controlled by a single four-channel DA-converter with a combined speed of 800 kSamp/sec and 16 bit resolution. **OUT A** is differential, so that a negative copy of the voltage is put between the - port and the shield. When used differentially **OUT A** will supply twice the value of the set voltage. Each port is driven by an operational amplifier with only one output range:

> ± 12 V , maximum current 70 mA.

9.4 Trigger Output

The trigger output ports are driven by active level shifter circuits for digital signals. The level shifter can be configured to 5 V, 3.3 V or 2.5 V voltage standards by an on-board jumper. The same jumper is used to set the voltage standard for all trigger input and output ports. Trigger output ports have 50 Ω output impedance and are capable of driving 50 Ω inputs at the other end.

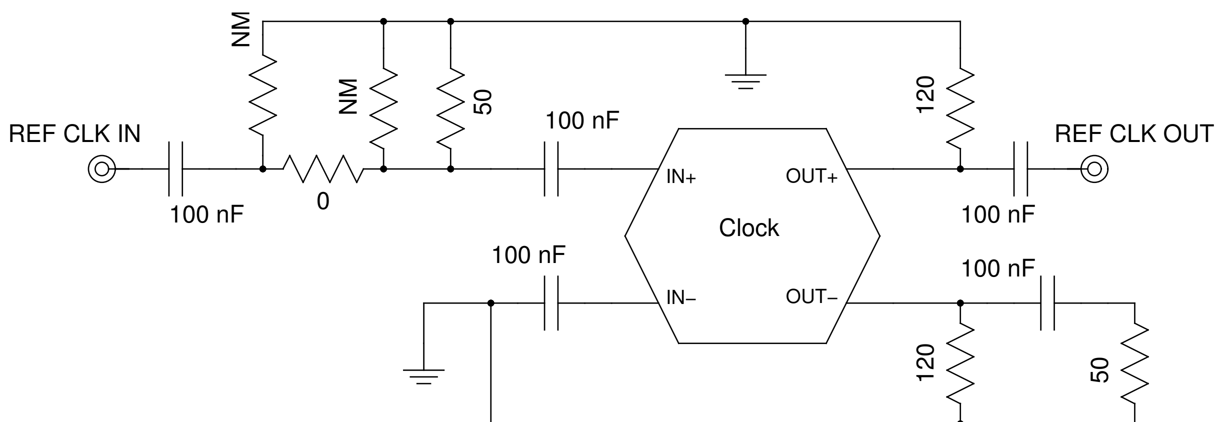
[1] when nodamp=False and term=True, the input impedance is 1kOhm, and the signal is divided by 20.

> **Caution:** When the both switches dc=True and bypass=True, both + and - inputs will be pulled up to 1 V.

9.5 Trigger Input

The trigger input ports are connected to active level shifter circuits for digital signals. The level shifter can be configured to 5 V, 3.3 V or 2.5 V voltage standards by an on-board jumper. The same jumper is used to set the voltage standard for all trigger input and output ports. The trigger input ports have high input impedance, which is common for trigger circuits. If you have trouble with reflections in long cables, use an T-connector with a 50 Ω terminator at the trigger input port. If you use the T-connector solution, remember to verify that the trigger output at the other end can drive the 50 Ohm load.

9.6 Clock Reference



The reference clock ports are AC-coupled. The voltage to the REF CLK IN port should be 0.25 to 2.4 Vpp. The type of the REF CLK OUT signal can be modified from the software, see hardware.Hardware.set_clkout_type(). To set the frequecy of the REF CLK OUT port, see hardware.Hardware.set_clkout_divisor(). To set up the REF CLK IN to lock to an external 10 MHz input, see hardware.Hardware.set_clkref_external_10MHz(). To set the frequency that the REF CLK IN should lock to, see functions hardware.Hardware.set_clkin_parameters().

CHAPTER

TEN

ADVANCED HARDWARE AND SOFTWARE TOPICS

10.1 Calibration

A calibration is used to convert quantities between physical units and the digital units of the AD- and DA-converters. The MLA comes with a factory calibration for converting to voltage units. However, in a specific measurement the user may want to include the gain of a pre-amplifier in the calibration, or some other calibration constant which converts to some other physical quantity that is linear in the digital units, for example displacement, magnetic field or current. In these cases the user can create their own custom calibration file with the proper calibration constants and units so the quantity of interest can be read off directly in the software.

The calibration file uses the well-established **.ini** file format (see for example <http://wikipedia.org/wiki/INI_file>). Multiple calibration files can coexist in the system, but only one is active.

A calibration file can be stored in either of the two following places:

- The **settings\calibration** subfolder of the *user folder*.
- The MLA hardware unit itself. Calibration files stored here will automatically be copied to the **settings\calibration** folder every time the MLA software starts and connects to the MLA. These calibrations will follow the MLA when it is connected to a new computer. To copy your own custom calibration file to the MLA, see the function `hardware.Hardware.upload_calibration()`.

Three different calibrations are delivered with the MLA:

- `factory.ini` – a calibration made at the factory, specific for the particular MLA unit. This calibration file is stored in the MLA and activated by default.
- `native_digital.ini` – Select this calibration if you want to use raw digital values in the AD- and DA-converters. This calibration file will be written to the **settings\calibration** folder every time the software starts.
- `generic.ini` – A standard calibration that should be quite close for unmodified MLA boxes. This calibration file will be written to the **settings\calibration** folder every time the software starts.

10.1.1 changing the active calibration

The active calibration is most easily changed from the *MLA GUI*. From the `Configuration` pull-down menu, choose `Calibration` and select your calibration. The active calibration is checked in the menu. The name of this active file is stored in the MLA and it will be recalled every time the software connects to the MLA. The ability of the MLA to remember the active calibration can be overridden so that the software will remember the active calibration. By setting the key **override_calibration = yes** in the file *mla_config.ini*, the software will load the calibration file specified by the key **calibration_filename** in this same file. This override functionality can be useful if the MLA is used in two or more different measurement setups, each with a separate computer.

The active calibration may also be set from a script or python terminal by calling the function `mla.convert.set_calfile()`.

10.1.2 creating a new calibration

Do not edit any of the existing calibrations. Such edits will be overwritten when the software starts. A new calibration should created by copying an existing calibration to a new file name and editing the contents. To copy the new calibration file to the MLA see hardware.Hardware.upload_calibration().

There is also a guided, text-based, process for creating new calibrations. It is invoked by calling the function mla.run_calibration_sequence(). If the calibration sequence is finished the new calibration will be stored in the MLA and activated as the default calibration. The MLA GUI can be quickly set up for this sequence by running the built-in script **calibrate.py**.

10.1.3 calibration file format

The conversion between physical and digital units uses parameters **offset** and **range** according to the following formula:

$\text{digital} = \text{physical} * \text{DIGITAL_RANGE}/\text{range} + \text{offset}$

and equivalently

$\text{physical} = (\text{digital-offset}) * \text{range}/\text{DIGITAL_RANGE}.$

where DIGITAL_RANGE is an internal constant of the software containing the highest possible (unsigned) number for the corresponding digital quantity (eg. 65535 for a 16-bit value).

This means that the **range** parameter corresponds to the maximum peak-to-peak amplitude of the physical quantity. And the **offset** parameter corresponds to the digital offset correction required to make a digital zero correspond to a physical zero.

Each port has a **range** and **offset** parameter, defined in the calibration file. For ports IN 1-IN 4 the parameters are called **AD_offset** and **AD_range**. For ports OUT 1 and OUT 2 the parameters are called **DA_offset** and **DA_range**. The ports OUT 1 and OUT 2 have an additional common mode voltage controlled by the AUXDAC (see *signal outputs*) which is calibrated with the parameters **auxdac_range** and **auxdac_offset**. For ports OUT A, OUT B, OUT C, OUT D the parameters are called **slowDA_offset** and **slowDA_range**.

10.2 Ethernet communication

The MLA supports gigabit Ethernet communication. Four different ports are used:

- **4250 (TCP) – Messages.** This port is used to send commands and queries to the MLA. The reply to the queries are also sent on this port. This is the only port where data is sent in both directions.
- **4251 (UDP) – Debug.** This port is used by server to send debug messages. Listening to this port is optional.
- **4252 (TCP) – Time data.** This port is used to send data sampled in the time domain. The data format is int16. There are no headers or meta data in this stream.
- **4253 (TCP) – Lockin data.** This port is used to send lockin data.

10.2.1 setting the IP number

The IP number of the MLA can be changed using the IMP MLA software from a computer which is able to connect to the MLA.

- Start the IMP MLA Software and wait until connection is established.
- In the *MLA GUI*, open the *Python shell panel*.
- Type the Python command **setup.set_ip_address(address='NEW_IP', current_ip='CURRENT_IP')**. You can configure the MLA to obtain automatic network settings via DHCP by **address='dhcp'**. You can also set netmask and gateway by their corresponding optional arguments, e.g. **netmask='255.255.0.0'**, **gateway=192.168.0.1**.

- Exit the software.
- Edit *mla_config.ini* and add **mla_ip = NEW_IP** under the heading **[COMMUNICATION]**.
- Configure the controlling computer such that it is able to reach the MLA at its new IP through its network.
- Cycle the power of the MLA and restart the software.

> **Caution:** The MLA is not designed to be exposed directly to the internet, please ensure it is only accessible through local networks.

10.2.2 resetting the IP number

The method above requires knowledge of the MLAs current IP address. If you have lost the current IP address and therefore can not connect to the MLA, the following trick allow you to "unbrick" the MLA. The MLA will have its original default IP address 192.168.42.50 for a few seconds during bootup.

- Turn off the MLA and quit the MLA software.
- Connect the MLA directly to an ethernet port on the computer and set the computer IP according to the *Quick Start Guide* instructions
- Edit *mla_config.ini* and make sure it says **mla_ip = 192.168.42.50** under the heading **[COMMUNICATION]** and **unbrick=True** under the heading **[FPGA]**.
- Start the MLA Software, but do not power up the MLA.
- When the software is repeatedly trying to ping, power up the MLA.

10.2.3 Message format

<Message length><Message ID><Payload>

- Message length (uint32) in bytes. The length includes its own four bytes.
- Message ID (unit32).
- Payload. The payload can be uint32, int32 or a string depending on the message ID.

10.2.4 Streams

Before a message can be sent to the MLA it must be packaged into a *stream*. A stream can contain one or more messages. The MLA will not execute a command until an entire stream has been received. When a full stream has been received, the MLA will execute all commands in the stream sequentially with no delay between each command. A stream starts with characters "star" followed by the length of the stream (not including the "star" or the length word). Then each message is appended directly after each other.

10.2.5 Lockin data packet

The lockin data is transmitted in packets where each packet contains the multifrequency data for a single measurement time window. The format of the lockin data packets is described here.

<header><cfg_cnt><trig1_cnt><trig2_cnt><data_cnt><trig_pos><I_0><Q_0><I_1><Q_1><I_2><Q_2>...<I_{N-1}><Q_{N-1}>

<header> (4*char) – Indicates start of a packet (the characters are "IMP1").

<cfg_cnt> (uint32) – A counter for how many times the configuration has been changed. Useful to determine which frame belongs to which configuration when rapidly changing configuration parameters, for example in a frequency sweep.

<trig1_cnt> (uint32) – Counter for how many trigger events received at port TRIG IN 1. When scanning an image, the line trigger output from the scanner is connected to TRIG IN 1.

<trig2_cnt> (uint32) – Counter for how many trigger events received at port TRIG IN 2. When scanning an image, the frame trigger output from the scanner is connected to TRIG IN 2.

<data_cnt> (uint32) – Counter for how many measurement time windows that lockin calculation has been active.

<trig_pos> (uint32) – Tells at which sample in the time window the trigger event occurred on trigger port TRIG IN 1.

<I_i> (int64) – Demodulated value of I quadrature at tone with index i. N is the number of tones in the MLA. Data is in raw format, not divided by number of samples, and phase is relative to drive phase at this tone.

<Q_i> (int64) – Demodulated value of Q quadrature at tone with index i. N is the number of tones in the MLA. Data is in raw format, not decided by number of samples, and phase is relative to drive phase at this tone.

10.3 Direct Memory Access (DMA)

Direct Memory Access (DMA) allows data to be copied from the MLA™ RAM memory, to the FPGA logic, and vise-versa. The MLA™ has DMA channels for fast transfer of big data blocks between the FPGA logic that controls the hardware, and the MLA RAM memory that can buffer large amounts of data (768 MByte) for communication with computer.

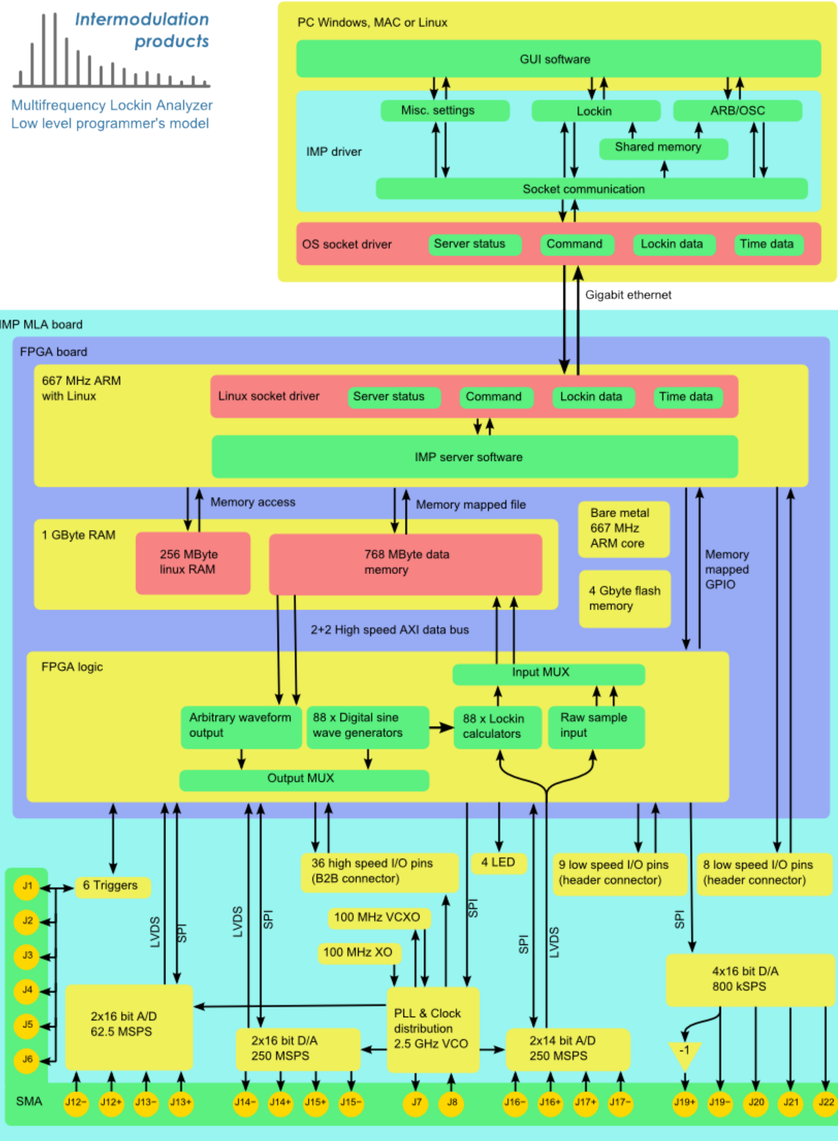
There are two DMA channels for transferring waveform data from the RAM to the DA-converters. This means that it is possible to simultaneously generate arbitrary waveforms on both of the fast DA-converters. The cosine waveforms used for the lockin measurements are generated in real time in the FPGA logic and thus they do not need any DMA resource.

There are also two DMA channels for transferring measurement data from the FPGA logic to the RAM memory. Measurement data can be either raw time samples, down-sampled time samples, or lockin data. This means that it is possible to simultaneously measure two time data channels, or one time data channel and one lockin data channel. By default, DMA channel 0 is used for time samples, and DMA channel 1 is used for lockin data.

To use the DMA resources differently than default, set the *dma_channel* parameter in some function calls such as osc.Osc.set_downsampling(), osc.Osc.start_time_stream(), lockin.Lockin.start_lockin() etc.

In the start-up of the MLA sequence, each of the four DMA channels are associated with 128 megabytes of RAM memory on the MLA. This can be reconfigured with the function hardware.Hardware.configure_memory(), see for example the built-in script check_noise_floor.py where 760 megabytes are associated with the DMA for ad converter 0 to allow a very long and continuous time trace to be measured at 250 Msamples/sec.

10.4 Low-level programmer's model



Intermodulation products
Multifrequency Lockin Analyzer
Low level programmer's model
PC Windows, MAC or Linux
GUI software
Misc. settings
Lockin
ARB/OSC
IMP driver
Shared memory
Socket communication
OS socket driver
Server status
Command
Lockin data
Time data
Gigabit ethernet
IMP MLA board
FPGA board
667 MHz ARM with Linux
Linux socket driver
Server status
Command
Lockin data
Time data
IMP server software
Memory access
Memory mapped file
Bare metal 667 MHz ARM core
1 GByte RAM
256 MByte linux RAM
768 MByte data memory
4 Gbyte flash memory
Memory mapped GPIO
2+2 High speed AXI data bus
FPGA logic
Input MUX
Arbitrary waveform output
88 x Digital sine wave generators
88 x Lockin calculators
Raw sample input
Output MUX
6 Triggers
36 high speed I/O pins (B2B connector)
4 LED
9 low speed I/O pins (header connector)
8 low speed I/O pins (header connector)
LVDS
SPI
100 MHz VCXO
100 MHz XO
4x16 bit D/A 800 kSPS
2x16 bit A/D 62.5 MSPS
PLL & Clock distribution 2.5 GHz VCO
2x16 bit D/A 250 MSPS
2x14 bit A/D 250 MSPS
-1
SMA
J1
J2
J3
J4
J5
J6
J12−
J12+
J13−
J13+
J14−
J14+
J15+
J15−
J7
J8
J16−
J16+
J17+
J17−
J19+
J19−
J20
J21
J22

CHAPTER

ELEVEN

REFERENCES

CHAPTER

TWELVE

GLOSSARY

Multi-frequency lockin measurement and intermodulation spectroscopy are developing fields and the language used to describe the concepts changes as our understanding becomes more refined. Mathematical and programing symbols are part of this language. We provide this glossary to establish definitions of the terms used in this manual.

12.1 Table of symbols used in this manual

Term	Programming	Math
Base tone	Df = 1/T	$\Delta f = 1/T$
Waveform period	T = 1/Df	$T = 1/\Delta f$
Measurement time	Tm = 1/df	$T_m = 1/\delta f$
Measurement bandwidth	df = 1/Tm	$\delta f = 1/T_m$
Tone index	idx	i
Tone-integer array	narray	n_i
Frequency array	freqs = narray*df	$f_i = n_i * df$

12.2 amplitude

The word amplitude is used in different ways and its exact definition depends on the context. Consider a signal consisting of one pure tone.

$V(t) = A\cos(\omega t + \phi)$.

For this single-frequency signal, the amplitude A is just the peak value of the signal. This amplitude is related to the two quadratures I and Q by the following expression,

$A = 2\sqrt{I^2 + Q^2}$

as described in the section on *lockin measurement*. We may also refer to the peak-to-peak amplitude of the signal,

$V_{\mathrm{pp}} = 2A$

or the root-mean-square (rms) amplitude of the signal,

$V_{\mathrm{RMS}} = \sqrt{\frac{1}{T_m}\int_0^{T_m} V^2(t)dt} = \frac{1}{\sqrt{2}}A$

when the *measurement time window* T_m is an integer multiple of the period $T = 2\pi/\omega$.

The table below summarizes how these various amplitudes are related.

Name	Symbol	Mathematical relationship
amplitude	A	A
quadrature amplitude	Aquad	$\sqrt{I^2 + Q^2} = A/2$ *
peak-to-peak amplitude	Vpp	$V_{pp} = 2A$
root-mean-square amplitude	Vrms	$V_{rms} = \frac{1}{\sqrt{2}} A$ *

* $= A$ at $\omega = 0$ or DC

Note that when we build a multifrequency waveform consisting of a superposition (sum) of N sinusoidal oscillations,

$V(t) = \sum_{i=0}^{N} A_i \cos(\omega_i t + \phi_i)$

it is not trivial to determine the peak, or maximum voltage of the signal. Each component oscillates with a different frequency and arbitrary phase and one can not generally say how all signals superpose to create the resultant signal.

12.3 base tone

The frequency of the base tone is the inverse of the waveform period. Any periodic waveform is represented on a comb in the frequency-domain, where all tones in the comb have frequencies that are integer multiples of the frequency of the base tone. The base tone is the greatest common divisor of all frequencies in the comb.

programing symbol - Df = 1/T

math symbol $\Delta f = 1/T$

12.4 frequency comb

We use frequency comb to mean **any** superposition of sine and cosine oscillations from a discrete set frequencies which are integer multiples of a *base tone* Δf.

$f_i = k_i \Delta f$,

where k_i are a set of integers.

The term 'comb' is used because the spectrum of all such tones with equal amplitude, looks like the instrument used to straighten your hair. The Fourier components of a comb are referred to as *tones*.

In optics, frequency combs generated from nonlinear media often have an offset frequency.

$f_i = f_{\text{offset}} + k_i \Delta f$

The MLA™ digitally synthesis a comb having a maximum of 40 tones, with $f_{\text{offset}} = 0$, and measures the response these frequencies.

12.5 measurement time window

The time window over which Fourier sums, or Lockin calculations are preformed. For optimal Fourier analysis, all tones of a multifrequency waveform should have a period which is an integer fraction of the measurement time window. The measurement time is the inverse of the measurement bandwidth.

programing symbol: - Tm = 1/df

math symbol $T_m = 1/\delta f$

12.6 measurement bandwidth

The inverse of the time window. Noise power is inversely proportional to the measurement bandwidth.

programing symbol - df = 1/Tm

math symbol $\delta f = 1/T_m$

Do not use and depreciate: delta_f, Delta_f, Δf

12.7 pixel

A fundamental unit of multifrequency lockin data containing the quadrature amplitudes at all measured frequencies. Pixel can also refer to the particular measurement time window on which this lockin data was calculated. This term comes from scanning probe microscopy, where each mutifrequency lockin measurement corresponds to one image pixel.

12.8 tones

The components of a multifrequency waveform, or components of a frequency comb. Tones have frequency, amplitude and phase, or alternatively, frequency and two quadrature amplitudes. Sometimes tones are loosely referred to 'frequencies', but strictly speaking, the frequency is only one of three attributes of a tone.

The tones in the MLA™, are specified by a **tone index** $i = 0, 1, 2, \ldots, N-1$, where N is the number of tones in the MLA™. The **frequency array** maps the tone index to frequency.

programming symbols: freqs = narray * df

- freqs : np.array(dtype=float) [Hz] frequencies of tones.
- narray : np.array(dtype=int) [-] frequencies of tones as multiples of df.
- df : float [Hz] measurement bandwidth

math symbols: $f_i = n_i \delta f$

12.9 tuning

The process of choosing the frequency of all measurement and excitation tones, as well as the measurement bandwidth, such that all have an integer relation to one base tone. In mathematical terms, tuning enforces the orthogonality of the drive tones and all their intermodulation products, on the finite interval of time over which the signal is integrated.

12.10 units

The MLA™ is a digital instrument that works with discrete numbers. We use the word 'units' in this context to distinguish which number we refer to:

- PCU - phase counter units, integer between 0 and 2**42.
- ADU - analog to digital units, integer steps of AD converter.
- DAU - digital to analog units, integer steps of DA converter.

The documentation should give physical units in hard brackets e.g. [Hz], unless the quantity is obviously non-dimensional (i.e. unit-less), in which case one could also write [-].

12.11 waveform period

The period of a multifrequency signal, for example the period of the drive waveform or period of the response waveform.

programing symbol - T = 1/Df

math symbol $T = 1/\Delta f$

BIBLIOGRAPHY

[Dicke-1946] Measurement of Thermal Radiation at Microwave Frequencies. R. H. Dicke REv. Sci. Instrum. **17**, 268 (1946). Dicke_1946

[Platz-2008] Intermodulation atomic force microscopy. D. Platz, E. A. Tholen, D. Pessen and D. B. Haviland. Appl. Phys. Lett. **92**, 153106 (2008). Platz_2008

[Tholen-2011] The intermodulation lockin analyzer. E. A. Tholen, D. Platz, D. Forchheimer, V. Schuler, M. O. Tholen, C. Hutter and D. B. Haviland. Rev. Sci. Instr. **82**, 026109 (2011). Tholen_2011

PYTHON MODULE INDEX

m

mlaapi.impgui, 95
mlaapi.lockin_plotpanel, 97
mlaapi.mla_gui, 93
mlaapi.scopepanel, 96
mlaapi.scriptpanel, 94
mlaapi.scriptutils, 95