Hyperion

Release 0.6

Sep 21, 2021

Contents:

This is the documentation for the Hyperion project (which is based on Python For the Lab). It is the software to control devices in the Kuipers lab (Kuipers Lab) group at QN, TU Delft.

CHAPTER 1

Index for this documentation

1.1 About

The Hyperion package is designed to help you to build your application to control a complex experimental setup. It should be seen as a scaffold to build the instrumentation software you need for the unique application or experiment you are working on.

Hyperion is under continuous development and started at the Kuipers Lab at Technical University Delft. We made special efforts to solve the specific problems we encounter at the lab when developing software that automatizes repetitive tasks and allows researches to perform reproducible experiments.

1.2 Overview

Common framework to glue together different devices in order to control a complex experiment.

TO DO: Show examples on how it works

1.3 Features

Here we mention some of the features we find interesting to remark.

1.3.1 Logging

We have an advanced and easy to configure logging implementation that provides you full record of what the program is doing.

1.3.2 Easy connection

1.3.3 Lantz-compatibility

We use Lantz as a dependency and we made special efforts to be completely compatible.

1.3.4 'smart' scan

1.3.5 Master GUI

We developed what we call a Master GUI where you can easily add your instruments and experiments using the basic canvas we developed.

1.4 Hyperion Architecture

In this project we try to use general structure known as **MVC**: **Model**, **View**, **Controller**, used for websites. We take some ideas from there and put it together with some of our ideas.

In a nutshell, we use an onion principle to isolate the code that interacts with the instruments directly from the code that builds up the user interface.

1.4.1 General structure

The **controller** refers to the lowest level, where the actual communication with the devices happens. We use the language of the devices at this level.

The **instrument** is an intermediate layer where we abstract the specifics of the device into a more general mode. This helps to adapt to the view layer. It is the name we give to the model layer in the MVC, since it is more suitable for instrumentation projects.

Several instruments are condensed together to form an **experiment** where you can perform different measurements. In the folder examples you can find a few explanatory files that show how to use the complete package.

The **view** is all that concerns the graphical user interface (GUI) of the applications. It consist of files that build up the GUI and some extra python files to load all the modules needed to run it. Under the hook, this uses the classes from the lower levels. You can have a GUI for an instrument, that connects only to one device or a GUI for an experiment that connects to several devices. The design principle here is that any experiment and measurement should be independent of the view layer and thus can be run by just running code up to the experiment layer (not using the view layer). Then the view layer can be optional for users that require the graphical control of their experiment.

1.4.2 Important concepts

Controller:

Instrument:

Meta-Instrument:

Experiment:

View:

1.4.3 Detailed documentation

Here you can see the detailed documentation of all the objects in the package.

Controllers

Here we group all the controllers we use for the Hyperion project.

AOTF controller

This controller (aa_modd18012.py) supplies one class with several methods to communicate with the AOTF driver from AA optoelectronics model: 1MODD18012_0074

copyright 2020by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.aa.aa_modd18012.**AaModd18012**(*settings*)

controller class for the driver aa_mod18012 from AA optoelelectronics. This class has all the methods to communicate using serial.

NOTE: Our model has different ranges of frequency (see data sheet) Line 1 to 6: 82-151 MHz (this drives short wavelengths) Line 7 to 8: 68-82 MHz (this drives long wavelengths)

Parameters **settings** (*dict*) – this includes all the settings needed to connect to the device in question.

blanking(*state*, *mode*)

Define the blanking state. If True (False), all channels are on (off). It can be set to 'internal' or 'external', where external means that the modulation voltage of the channel will be used to define the channel output.

Parameters

- **state** (*logical*) – State of the blanking
- **mode** (*string*) – external or internal. external is used to follow TTL external modulation

check_channel(*channel*)

Method to check the key of the channel is correct

Parameters **channel** (*int*) – channel to use

Returns channel

Return type int

check_freq(*channel*, *value*)

Checks if the frequency asked is valid for the desired channel. Specific of our device. If it is not, it gives the right value.

Parameters

- **channel** (*int*) – channel to use (can be from 1 to 8 inclusive)
- **value** (*float*) – Frequency value in MHz. If 0 is given, the DEFAULTS['frequency'] value is used for the corresponding channel.

Returns The channel and value back unless an error.

Return type int, new_channel

check_power(*value*)
checks if the power is in the range supported by the device. Range = (0 , 22) dBm. Gives an error if value is not in the range

Parameters **value** (*float*) – power value in dBm

enable(*channel*, *state*)
Enable single channels.

Parameters

- **channel** – channel to use (can be from 1 to 8 inclusive)
- **state** (*logical*) – true for on and false for off

Type channel: int

Returns state

Return type logical

finalize()
Closes the connection to the device

get_states()
Gets the status of all the channels

Returns message from the driver describing all channel state

Return type str

initialize()
Initialize the device. It actually connects to the device.

query(*message*)
Writes and Reads the answer from the device. Basically this method is a write followed by a read with a 10ms wait in the middle.

Parameters **message** (*string*) – Message to be passed to the serial port.

Returns The reply from the device.

Return type string

read()
Reads message from the device. It reads until the buffer is clean.

Returns The messages in the buffer of the device. It removes the end of line characters.

Return type string

set_all(*channel*, *freq*, *power*, *state*, *mode*)
Sets Frequency, power, and state of channel.

Parameters

- **channel** (*int*) – channel to use (can be from 1 to 8 inclusive)
- **freq** (*float*) – Frequency value in MHz (range depends on the channel)
- **power** (*float*) – Power to set in dBm (0 to 22)
- **state** (*logical*) – true for on and false for off
- **mode** (*string*) – ‘internal’ or ‘external’

set_frequency(*channel*, *freq*)
This function sets RF frequency for a given channel. The device has 8 channels. Channels 1-6 work in the range 82-151 MHz Channels 7-8 work in the range 68-82 MHz

Parameters

- **channel** (*int*) – channel to set the frequency.
- **freq** (*float*) – Frequency to set in MHz (it has accepted ranges that depends on the channel)

set_operating_mode(*channel*, *mode*)
Select the operating mode. Can be internal or external.

Parameters

- **channel** (*int*) – channel number
- **mode** (*str*) – 'internal' or 'external'

set_powerdb(*channel*, *value*)
Power for a given channel (in dBm). Range: (0,22) dBm

Parameters

- **channel** (*int*) – channel to use
- **value** (*float*) – power value in dBm

store()
stores in the internal memory the values

write(*message*)
Sends the message to the device.

Parameters **message** (*string*) – message to send to the device.

Returns the response from the device.

Return type string

class hyperion.controller.aa.aa_modd18012.**AaModd18012Dummy**(*settings={'dummy': True, 'port': 'COM00'}*)
This is the dummy controller for the AaModd18012. The idea is to load this class instead of the real one to do testing of higher level functions without the need of the real device to be connected or working.

The logic is that this dummy device will respond as the real device would, with the correct type and size of information is expected.

This class inherits from the real device and the idea is to re-write only the init, the write and the read, so all the other functions remain the same and functioning.

The specific way to achieve this will be different for every device, so it has to be done separately.

To do so, we use a yaml file that tells the dummy class what are the properties of the device. For example, one property for the LCC25 is voltage1, which is the voltage for channel 1. Then from this you can build 2 commands: voltage1? to ask what is the value and voltage1=1 to set it to the value 1. So we build a command list using the CHAR ? and = for each of this properties.

load_properties()
This method loads a yaml file with a dictionary with the available properties for the AaModd18012 and some defaults values. This dictionary is saved in properties and will be modified when a variable is writen, so the dummy device will respond with the previously set value.

read()
Dummy read. Reads the response buffer

write(*msg*)
Dummy write. It will compare the msg with the COMMANDS

Parameters **msg** (*str*) – Message to write

Agilent 33522A controller

This is the controller class for the Agilent 33522A function generator. Based on pyvisa to send commands to the USB.

copyright 2020by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.agilent.agilent33522A.**Agilent33522A**(*settings*)
Agilent 33522A arbitrary waveform generator, 30MHz, 2 channels. It takes a dictionary that passes the settings needed. In this case it needs

instrument_id : '8967' # instrument id for the device you have dummy : False

Parameters **settings** (*dict*) – a dictionary for the settings you need to send to the device

check_channel(*channel*)
Function to check if the channel is present in the system.

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns The number of the channel

Return type string

enable_output(*channel*, *state*)
Enable the output of channel 1 or 2

Parameters

- **channel** (*int*) – To activate or deactivate
- **state** (*logical*) – logical state. True sets the output on and false off.

enable_voltage_limits(*channel*, *state*)

This function enables the limits for the maximum and minimum voltage output that can be generated by each channel. Those values are set with the method set_voltage_limits(channel, high, low).

This function enables this setting by putting state = true and disables it by putting state = false. When enable, setting a voltage outside the permitted values will give an error (not in python, in the device)

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **state** (*logical*) – True to turn on, False to turn off

finalize()
This methods closes the visa connection

get_enable_output(*channel*)
Get the status of the output. 0 is off, 1 is on.

Parameters **channel** (*int*) – can be 1 or 2 for this model

Returns Status of the output

Return type logical

get_frequency(*channel*)
This functions reads the frequency output for the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns Frequency value currently in the device in Hz

Return type string

get_state_voltage_limits(*channel*)
This function generator can set a minimum and maximum voltage value that will not be exceeded to protect the device that is being feed.

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns voltage limits

Return type string

get_system_error()
This functions returns the error message

Returns error message

Return type string

get_voltage(*channel*)
This functions gets the voltage in the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns current voltage Vpp in the device

Return type string

get_voltage_high(*channel*)
This functions sets the high voltage to the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns High voltage value

Return type string

get_voltage_limits(*channel*)
Gets the set values for the voltage limits, high and low.

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns array with [high_value, low_value] in volts

Return type array of strings

get_voltage_limits_state(*channel*)
Checks the status of the voltage limits. It can be on or off

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns voltage limit state (logical)

Return type string

get_voltage_low(*channel*)
This functions sets the low voltage to the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns Low voltage set on the device

Return type string

get_voltage_offset(*channel*)
This functions gets the DC offset voltage to the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns current offset in the device

Return type string

get_waveform(*channel*)
Get the function set for the output. The available functions are stored at FUNCTIONS = ['SIN','SQU','TRI','RAMP','PULS','PRBS','NOIS','ARB','DC']

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model the expected output is on of the items in FUNCTIONS

Returns type of function in use

Return type string

idn()
Ask the device for its identification

Returns identification of the fun gen

Return type string

initialize()
This method opens the communication with the device.

query(*msg*)
Sequential read and write :param msg: message to write to the device :type msg: string

read()
Read buffer

set_frequency(*channel*, *freq*)
This functions sets the frequency output for the channel

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **freq** (*float*) – desired frequency in Hz

set_voltage(*channel*, *voltage*)
This functions sets the Vpp voltage to the channel

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **voltage** (*float*) – voltage value for the high voltage in volts (with sign)

set_voltage_high(*channel*, *voltage*)
This functions sets the high voltage to the channel

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **voltage** (*float*) – voltage value for the high voltage in volts (with sign)

set_voltage_limits(*channel*, *high*, *low*)
Set a limit to the output values

For this function generator you can set a minimum and maximum voltage value that will not be exceeded to protect the device that is being feed.

NOTE: setting the values does not activate this feature. To enable/disable it use the function enable_voltage_limits(channel,state)

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **high** (*float*) – High maximum voltage value for the output in Volts
- **low** (*float*) – High maximum voltage value for the output in VOlts

set_voltage_low(*channel*, *voltage*)
This functions sets the low voltage (in volts) to the channel.

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **voltage** (*float*) – voltage value for the low voltage in volts (with sign)

set_voltage_offset(*channel*, *voltage*)
This functions sets the DC offset voltage to the channel

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **voltage** (*float*) – voltage value for the DC offset voltage in volts (with sign)

set_waveform(*channel*, *fun*)
Get the function set for the output. The available functions are stored at FUNCTIONS = ['SIN','SQU','TRI','RAMP','PULS','PRBS','NOIS','ARB','DC']

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **fun** (*string*) – One of the functions defined in FUNCTIONS. Ex: 'SIN"

write(*msg*)
Write in the device buffer :param msg: message to write to the device :type msg: string

class hyperion.controller.agilent.agilent33522A.**Agilent33522ADummy**(*settings*)
This is the dummy controller for the Agilent33522A.

The idea is to load this class instead of the real one to do testing of higher level functions without the need of the real device to be connected or working.

The logic is that this dummy device will respond as the real device would, with the correct type and size of information is expected.

This class inherits from the real device and the idea is to re-write only the init, the write and the read, so all the other functions remain the same and functioning.

The specific way to achieve this will be different for every device, so it has to be done separately.

To do so, we use a yaml file that tells the dummy class what are the properties of the device. For example, one property for the LCC25 is voltage1, which is the voltage for channel 1. Then from this you can build 2 commands: voltage1? to ask what is the value and voltage1=1 to set it to the value 1. So we build a command list using the CHAR ? and = for each of this properties.

load_properties()
This method loads a yaml file with a dictionary with the available properties for the LCC25 and some defaults values. This dictionary is saved in properties and will be modified when a variable is writen, so the dummy device will respond with the previously set value.

read()
Dummy read. Reads the response buffer

write(*msg*)
Dummy write. It will compare the msg with the COMMANDS

Parameters **msg** (*str*) – Message to write

ANC350 Attocube Controller

This is the controller level of the position ANC350 from Attocube (in the Montana)

This code is strongly based and using PyANC350, which was written by Rob Heath; rob@robheath.me.uk; 24-Feb-2015; It was taken from github in August 2019 by Irina Komen and made to work with Hyperion

Description from Rob Heath: PyANC350 is a control scheme suitable for the Python coding style for the attocube ANC350 closed-loop position system.

It implements ANC350lib, which in turn depends on anc350v2.dll which is provided by attocube in the ANC350_DLL folders on the driver disc. This in turn requires nhconnect.dll and libusb0.dll. Place all of these in the same folder as this module (and that of ANC350lib).

Unlike ANC350lib which is effectively a re-imagining of the C++ header, PyANC350 is intended to behave as one might expect Python to. This means: returning values; behaving as an object.

At present this only addresses the first ANC350 connected to the machine.

Usage:

1. instantiate Positioner() class to begin, eg. pos = Positioner().
2. methods from the ANC350v2 documentation are implemented such that function PositionerGetPosition(handle, axis, &pos) becomes position = pos.getPosition(axis), PositionerCapMeasure(handle,axis,&cap) becomes cap = pos.capMeasure(axis), and so on. Return code handling is within ANC350lib.
3. bitmask() and debitmask() functions have been added for convenience when using certain functions (e.g. getStatus,moveAbsoluteSync)

4. for tidiness remember to Positioner.close() when finished!

Important NOTE:

This code assumes that the following files are stored in this folder (hyperion/controller/attocube/)

- anc350v2.dll
- anc350v2.lib
- libusb0.dll

And the (actor) calibration files:

- ANPx101-A3-1079.aps
- ANPx101-A3-1087.aps
- ANPz102-F8-393.aps

You should get these files from the manufacturer (except for libusb0.dll).

class hyperion.controller.attocube.anc350.**Anc350**(*settings*)
Class for the ANC350 controller

Parameters **settings** (*dict*) – this includes all the settings needed to connect to the device in question, in this case just dummy.

acInEnable(*axis*, *state*)

UNUSED
Activates/deactivates AC input of addressed axis; only applicable for dither axes.

amplitude(*axis*, *amp*)

Set the amplitude setpoint of the Stepper in mV.
You need to set the amplitude, max 60V.
At room temperature you need 30V for x and y and 40V for z.
At low temperature that is higher, 40V or even 50V.
Higher amplitude influences step size though.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **amp** (*integer*) – amplitude to be set to the Stepper in mV, between 0 and 60V; needs to be an integer!

amplitudeControl(*axis*, *mode*)

Selects the type of amplitude control in the Stepper.
The amplitude is controlled by the positioner to hold the value constant determined by the selected type of amplitude control.
I thought for closed loop it needs to be set in Step Width mode, nr. 2.
However, that gives issues, since sometimes the amplitude is not high enough to make the thing move at all.
So Amplitude control mode, nr. 1, seems better.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **mode** (*integer*) – Mode takes values 0: speed, 1: amplitude, 2: step size

bandwidthLimitEnable(*axis*, *state*)

UNUSED
activates/deactivates the bandwidth limiter of the addressed axis. only applicable for scanner axes.

capMeasure(*axis*)

Determines the capacitance of the piezo addressed by axis.
Pay attention: the 0 that you give to the ANC350lib.Int32 is the first Attocube device; not the first of the 6 positioners.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers and 3 to 5 for scanners

Returns capacitance value in mF

check()
Determines number of connected attocube controller devices and their respective hardware IDs.

clearStopDetection(*axis*)

UNUSED
when .setStopDetectionSticky() is enabled, this clears the stop detection status.

connect()

Establishes connection to the first attocube device found.
Pay attention: the 0 that you give to the ANC350lib.Int32 is the first attocube device; not the first of the 6 positioners.

dcInEnable(*axis*, *state*)

UNUSED
Activates/deactivates DC input of addressed axis; only applicable for scanner/dither axes.

dcLevel(*axis*, *dclev*)
Sets the dc level of selected scanner (or maybe stepper, if you want).

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers and 3 to 5 for scanners
- **dclev** (*integer*) – DC level in mV; needs to be an integer!

dutyCycleEnable(*state*)
UNUSED controls duty cycle mode.

dutyCycleOffTime(*value*)
UNUSED sets duty cycle off time.

dutyCyclePeriod(*value*)
UNUSED sets duty cycle period.

externalStepBkwInput(*axis*, *input_trigger*)

UNUSED
configures external step trigger input for selected axis. a trigger on this input results in a backwards single step.
input_trigger: 0 disabled, 1-6 input trigger.

externalStepFwdInput(*axis*, *input_trigger*)

UNUSED
configures external step trigger input for selected axis.
a trigger on this input results in a forward single step.
input_trigger: 0 disabled, 1-6 input trigger.

externalStepInputEdge(*axis*, *edge*)

UNUSED
configures edge sensitivity of external step trigger input for selected axis.
edge: 0 rising, 1 falling.

finalize()
Closes connection to ANC350 device.

frequency(*axis*, *freq*)

Sets the frequency of selected stepper axis.
Higher means more noise and faster (= less precise?).

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **freq** (*integer*) – frequency in Hz, from 1Hz to 2kHz; needs to be an integer!

getAcInEnable(*axis*)

UNUSED
determines status of ac input of addressed axis. only applicable for dither axes.

getAmplitude(*axis*)

Determines the actual amplitude.
In case of standstill of the actor this is the amplitude setpoint.
In case of movement the amplitude set by amplitude control is determined.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

Returns measured amplitude in mV

getBandwidthLimitEnable(*axis*)

UNUSED
determines status of bandwidth limiter of addressed axis. only applicable for scanner axes.

getDcInEnable(*axis*)

UNUSED
determines status of dc input of addressed axis. only applicable for scanner/dither axes.

getDcLevel(*axis*)

Determines the status actual DC level on the scanner (or stepper).
Again: the 0 is for the number of controller units, not the 6 positioners.

Parameters **axis** (*integer*) – axis number from 3 to 5 for scanners

Returns measured DC level in mV

getFrequency(*axis*)
Determines the frequency on the stepper.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

Returns measured frequency in Hz

getIntEnable(*axis*)
Determines status of internal signal generation of addressed axis. only applicable for scanner/dither axes.

Parameters **axis** (*integer*) – axis number from 3 to 5 for scanners

Returns True if the INT mode is selected, False if not

getPosition(*axis*)

Determines actual position of addressed stepper axis.
Pay attention: the sensor resolution is specified for 200nm.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

Returns position in nm

getReference(*axis*)
UNUSED determines distance of reference mark to origin.

getReferenceRotCount(*axis*)
UNUSED determines actual position of addressed axis.

getRotCount(*axis*)
UNUSED determines actual number of rotations in case of rotary actuator.

getSpeed(*axis*)

Determines the actual speed.
In case of standstill of this actor this is the calculated speed resultingfrom amplitude setpoint, frequency, and motor parameters.
In case of movement this is measured speed.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

Returns speed in nm/s

getStatus(*axis*)

Determines the status of the selected axis.

It is not clear whether it also works for the scanner, or only for the stepper.
result: bit0 (moving) (+1), bit1 (stop detected) (+2), bit2 (sensor error) (+4), bit3 (sensor disconnected) (+8).

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers and 3 to 5 for scanners

Returns 1: moving, 2: stop detected, 4: sensor error, 8: sensor disconnected

getStepwidth(*axis*)

Determines the step width.
In case of standstill of the motor this is the calculated step width resulting from amplitude setpoint, frequency, and motor parameters.
In case of movement this is measured step width.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

Returns stepwidth in nm

initialize()

Initializes the controller.
Checks for attocube controller units and connects to it.
Pay attention: there are 6 positioners, but only 1 controller; we connect to the 1.

intEnable(*axis*, *state*)
Activates/deactivates internal signal generation of addressed axis; only applicable for scanner/dither axes.

Parameters

- **axis** (*integer*) – axis number from 3 to 5 for scanners
- **state** (*bool*) – True is enabled, False is disabled

load(*axis*)

Loads a parameter file for actor configuration.
note: this requires a pointer to a char datatype.
the actor files are in this controller folder, their names are hard coded in the init.
note: Attocube called the up-down actor file ANPz, I called that axis YPiezo.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers and 3 to 5 for scanners

moveAbsolute(*axis*, *position*, *rotcount=0*)

Starts approach to absolute target position.
Previous movement will be stopped.
Rotcount optional argument, not in our case since we dont have rotation options.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **position** (*integer*) – absolute target position in nm; needs to be an integer!

- **rotcount** – optional argument position units are in 'unit of actor multiplied by 1000' (generally nanometres)

moveAbsoluteSync(*bitmask_of_axes*)

UNUSED
Starts the synchronous approach to absolute target position for selected axis.
Previous movement will be stopped.
Target position for each axis defined by .setTargetPos() takes a *bitmask* of axes!
Not clear what's the difference with moveAbsolute.

Parameters **bitmask_of_axes** –

moveContinuous(*axis*, *direction*)
Starts continuously positioning with set parameters for ampl and speed and amp control respectively.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **direction** (*integer*) – can be 0 (forward) or 1 (backward)

moveReference(*axis*)

UNUSED
Starts approach to reference position.
Previous movement will be stopped.
No idea whats the difference with moveRelative

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

moveRelative(*axis*, *position*, *rotcount=0*)

Starts approach to relative target position.
Previous movement will be stopped.
Rotcount optional argument, not in our case since we dont have rotation options.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **position** (*integer*) – relative target position in nm, can be both positive and negative; needs to be an integer!
- **rotcount** – optional argument position units are in 'unit of actor multiplied by 1000' (generally nanometres)

moveSingleStep(*axis*, *direction*)

Starts a one-step positioning, where the stepwidht is determined by the amplitude and frequency.
Previous movement will be stopped.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **direction** – can be 0 (forward) or 1 (backward)

quadratureAxis(*quadratureno*, *axis*)

UNUSED
selects the axis for use with this trigger in/out pair.
quadratureno: number of addressed quadrature unit (0-2).

quadratureInputPeriod(*quadratureno*, *period*)

UNUSED
selects the stepsize the controller executes when detecting a step on its input AB-signal.
quadratureno: number of addressed quadrature unit (0-2). period: stepsize in unit of actor * 1000.

quadratureOutputPeriod(*quadratureno*, *period*)

UNUSED
selects the position difference which causes a step on the output AB-signal.
quadratureno: number of addressed quadrature unit (0-2). period: period in unit of actor * 1000.

resetPosition(*axis*)

UNUSED
sets the origin to the actual position.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

sensorPowerGroupA(*state*)

UNUSED
switches power of sensor group A.
Sensor group A contains either the sensors of axis 1-3 or 1-2 dependent on hardware of controller.

sensorPowerGroupB(*state*)

UNUSED
switches power of sensor group B.
Sensor group B contains either the sensors of axis 4-6 or 3 dependent on hardware of controller.

setHardwareId(*hwid*)
UNUSED sets the hardware ID for the device (used to differentiate multiple devices).

setOutput(*axis*, *state*)

UNUSED
activates/deactivates the addressed axis.
no idea what that means, but sounds interesting.

Parameters

- **axis** –
- **state** –

setStopDetectionSticky(*axis*, *state*)

UNUSED
when enabled, an active stop detection status remains active until cleared manually by .clearStopDetection().
Is this what in Daisy is called hump detection? Than it might be useful.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **state** –

setTargetGround(*axis*, *state*)
UNUSED when enabled, actor voltage set to zero after closed-loop positioning finished.

setTargetPos(*axis*, *pos*, *rotcount=0*)
UNUSED sets target position for use with .moveAbsoluteSync().

singleCircleMode(*axis*, *state*)

UNUSED
switches single circle mode.
In case of activated single circle mode the number of rotations are ignored and the shortest way to target position is used. Only relevant for rotary actors.

staticAmplitude(*amp*)
UNUSED sets output voltage for resistive sensors.

stepCount(*axis*, *stps*)
UNUSED configures number of successive step scaused by external trigger or manual step request. steps = 1 to 65535.

stopApproach(*axis*)

Stops approaching target/relative/reference position.
DC level of affected axis after stopping depends on setting by .setTargetGround().
Dont know for sure whats the difference with stopMoving.

Parameters **axis** (*integer*) – axis number from 0 to 2 for steppers

stopDetection(*axis*, *state*)

UNUSED
switches stop detection on/off.
Is this what in Daisy is called hump detection? Than it might be useful.

Parameters

- **axis** (*integer*) – axis number from 0 to 2 for steppers
- **state** –

stopMoving(*axis*)

UNUSED
stops any positioning, DC level of affected axis is set to zero after stopping.

Parameters **axis** –

trigger(*triggerno*, *lowlevel*, *highlevel*)

UNUSED
sets the trigger thresholds for the external trigger.
triggerno is 0-5, lowlevel/highlevel in units of actor * 1000.

triggerAxis(*triggerno*, *axis*)
UNUSED selects the corresponding axis for the addressed trigger. triggerno is 0-5.

triggerEpsilon(*triggerno*, *epsilon*)
UNUSED sets the hysteresis of the external trigger. epsilon in units of actor * 1000.

triggerModeIn(*mode*)

UNUSED
selects the mode of the input trigger signals.
state: 0 disabled - inputs trigger nothing,
1 quadrature - three pairs of trigger in signals are used to accept AB-signals for relative positioning,
2 coarse - trigger in signals are used to generate coarse steps.

triggerModeOut(*mode*)

UNUSED
selects the mode of the output trigger signals.
state: 0 disabled - inputs trigger nothing,
1 position - the trigger outputs reacts to the defined position ranges with the selected polarity,
2 quadrature - three pairs of trigger out signals are used to signal relative movement as AB-signals, 3 IcHaus - the trigger out signals are used to output the internal position signal of num-sensors.

triggerPolarity(*triggerno*, *polarity*)
UNUSED

sets the polarity of the external trigger, triggerno: 0-5, polarity: 0 low active, 1 high active.

updateAbsolute(*axis*, *position*)

UNUSED
update s target position for a *running* approach.
function has lower performance impact on running approach compared to .moveAbsolute().
position units are in 'unit of actor multiplied by 1000' (generally nanometres).

Parameters

- **axis** –
- **position** –

class hyperion.controller.attocube.anc350.**Anc350Dummy**(*settings*)

hyperion.controller.attocube.anc350.**bitmask**(*input_array*)

takes an array or string and converts to integer bitmask.
reads from left to right e.g. 0100 = 2 not 4.

hyperion.controller.attocube.anc350.**debitmask**(*input_int*, *num_bits=False*)

takes a bitmask and returns a list of which bits are switched.
reads from left to right e.g. 2 = [0, 1] not [1, 0].

Base controller

This is a base class for a controller. The idea is to use this class as the parent class of any driver you are writing by hand. By doing so, you gain the possibility of using the context manner for the 'with' block and you how which basic methods you should write. We strongly recommend you also use the logging, so you can keep track of what is going on with your program at every step.

class hyperion.controller.base_controller.**BaseController**(*settings={}*)
General class for controller. Use it as parent of your (home-made) controller.

finalize()
This method closes the connection to the device. It is ran automatically if you use a with block

idn()
Identify command

initialize()
Starts the connection to the device.

Cobolt 08NLD

This is the controller for the Cobotl laser 08NLD

Based on the driver for laser 06-01 series by Vasco Tenner, available in lantz.drivers.laser.cobolt08NLD

copyright 2020by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.cobolt.cobolt08NLD.**Cobolt08NLD**(*resource_name*, *name=None*, ***kwargs*)
controller class for the driver COBOLT 08-NLD Series laser. This class has all the methods to communicate using serial.

analog_mod
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)

2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

analogli_mod

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

autostart

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

clear_fault = <lantz.core.action.Action object>

clear_fault_async(**args*, ***kwargs*)
(Async) Clear fault

ctl_mode
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

current_sp
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

digital_mod
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

enabled
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

enter_mod_mode = <lantz.core.action.Action object>

enter_mod_mode_async(**args*, ***kwargs*)
(Async) Enter modulation mode

idn
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.

6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

interlock

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

ksw_enabled

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

mod_mode

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

operating_hours

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

power

Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.

8. Finally the actual getter or setter is called.

power_sp
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

restart = <lantz.core.action.Action object>

restart_async(**args*, ***kwargs*)
(Async) Forces the laser on without checking if autostart is enabled.

status
Pimped Python property for interfacing with instruments. Can be used as a decorator.

Processors can registered for each arguments to modify their values before they are passed to the body of the method. Two standard processors are defined: *values* and *units* and others can be given as callables in the *get_funcs* parameter.

If a method contains multiple arguments, use the *item* method.

Feat has the following nested behaviors:

1. Feat: lantz specific modifiers: values, units, limits, procs, read_once)
2. LockProperty: locks the parent drive (for multi-threading apps)
3. ObservableProperty: emits a signal when the cached value has changed (via set/get)
4. SetCacheProperty: prevents unnecessary set operations by comparing the value in the cache
5. TransformProperty: transform values according to predefined rules.
6. LogProperty: log get and set operations
7. StatsProperty: record number of calls and timing stats for get/set/failed operations.
8. Finally the actual getter or setter is called.

Example controller

This is an example of a controller with a fake (invented) device. It should help to gide developers to create new controllers for real devices.

class hyperion.controller.example_controller.**ExampleController**(*settings*)
Example output device that does not connect to anything

amplitude
Gets the amplitude value.

Getter

Returns amplitude value in Volts

Return type float

Setter

Parameters **value** (*float*) – value for the amplitude to set in Volts

For example, to use the getter you can do the following ampl = this_controller.amplitude For example, using the setter looks like this: this_controller.amplitude = 5

finalize()
This method closes the connection to the device. It is ran automatically if you use a with block

idn()
Identify command

Returns identification for the device

Return type string

initialize()
Starts the connection to the device in port

query(*msg*)
writes into the device msg

Parameters **msg** (*string*) – command to write into the device port

read()
Fake read that returns always the value in the dictionary FAKE RESULTS.

Returns fake result

Return type string

write(*msg*)
Writes into the device :param msg: message to be written in the device port :type msg: string

class hyperion.controller.example_controller.**ExampleControllerDummy**(*settings*)
A dummy version of the Example Controller.

In essence we have the same methods and we re-write the query to answer something meaningful but without connecting to the real device.

query(*msg*)
writes into the device msg

Parameters **msg** (*string*) – command to write into the device port

Generic Serial Controller

This is a generic controller for Serial devices (like Arduino). It includes serial write and query methods. Higher level stuff could (should?) be done at Instrument level. Extra methods could however also be added to this controller, as long as they don't break the existing functionality.

This controller is however intended to be agnostic of what code/firmware/sketch is running on the device. I suggest to keep it that way and put device specific functionality in a dedicated instrument. This also means Dummy mode needs to be implemented mostly at Instrument level.

Development note: I've added the experimental decorator _wait_while_busy_and_after. If it turns out not to work desirably it can be removed: remove the application of the decorator on methods initialize, write and read_serial_buffer_in remove the decorator function itself remove self._busy and self._additional_timeout from the __init__

class hyperion.controller.generic.generic_serial_contr.**GenericSerialController**(*settings*)
Generic Serial Controller

Parameters **settings** (*dict*) – This includes all the settings needed to connect to the device in question.

finalize()
This method closes the connection to the device. It is ran automatically if you use a with block.

idn()
Identify command.

Returns identification for the device

Return type string

query(*message*)
Writes message in the device Serial buffer and Reads the response. Note, it clears the input buffer before sending out the query.

Parameters **message** (*str*) – command to send to the device

Returns list of responses received from the device

Return type list of strings

read_lines(*remove_leading_trailing_empty_line=True*)

Reads all lines the device has sent and returns list of strings. It interprets both

and combinations as a newline character.

param remove_leading_trailing_empty_line defaults to True

type remove_leading_trailing_empty_line bool

return list of lines received from the device

rtype list of strings

class hyperion.controller.generic.generic_serial_contr.**GenericSerialControllerDummy**(*settings*)
A dummy version of the Generic Serial Controller.

Note that because Generic Serial Controller is supposed to be device agnostic this dummy can't simulate every device. For devices using this Controller, simulation has to be done at Instrument level. This Dummy only stores write messages in a buffer and returns this buffer with the read_serial_buffer_in method.

All other methods are are kept from GenericSerialController.

finalize()
Finalizes Generic Serial Controller Dummy device.

initialize()
Initializes Generic Serial Controller Dummy device.

query(*message*)
Simulates query to dummy device. Clears the internal buffer. Performs a write, followed by a read_lines(). Note, it clears the input buffer before sending out the query.

Parameters **message** (*str*) – command to send to the device

Returns response from the device

Return type str

read_serial_buffer_in(*wait_for_termination_char=True*)
Simulates read from dummy device (returns internal buffer).

write(*message*)
Simulates write to dummy device (adds message to an internal buffer). :param message: message to write :type message: string

Hydraharp400 controller

This code is strongly based on the code from Colin Brosseau, licensed as BSD 3-Clause "New" or "Revised" License:

Copyright (c) 2017, Colin Brosseau All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The original code was taken on Fri Nov 10 15:36:48 2017 from Github (https://github.com/ColinBrosseau/Hydraharp400), changed by Irina Komen to work within hyperion.

class hyperion.controller.picoquant.hydraharp.**Hydraharp**(*config*)

Hydraharp 400 controller
Initializes communication with Hydraharp400 device.

Parameters

- **devidx** (*int*) – index of the device
- **mode** (*string*) – operation mode, can be: 'Histogram'(default), 'T2','T3','Continuous'

- **clock** (*string*) – source of the clock, can be: 'External'(default), 'Internal'

calibrate()
Calibrate the device; no calibration file needed, this is an internal function.

Parameters **devidx** (*int*) – Index of the device (default 0)

clear_histogram()
Clear histogram from memory

count_rate(*channel=0*)

Current count rate of the input channel.
Allow at least 100 ms after HH_Initialize or HH_SetSyncDivider to get a stable rate meter readings.
Similarly, wait at least 100 ms to get a new reading. This is the gate time of the counters.

Parameters **channel** (*int*) – input channel index; in our case 0 or 1

return count rate: measured counts per second on one of the channels

ctc_status
Acquisition time state.

Return status False: acquisition time still running; True: acquisition time has ended

error_string
Error messages.

finalize()
Closes and releases the device for use by other programs.

flags
Use the predefined bit mask values in hhdefin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.

hardware_info
Information about the device.

histogram(*channel=0*, *clear=True*)

Histogram of channel.
Have to use this one only after starting a measurement!
The histogram is always taken between one of the input channels and the sync channel.
To perform start-stop measurements, connect one of the photon detectors to the sync channel.

Parameters

- **channel** (*int*) – input channel index; in our case 0 or 1
- **clear** (*bool*) – denotes the action upon completing the reading process; False keeps the histogram in the acquisition buffer; True clears the buffer

Return histogram array with the histogram data; size is determined by histogram_length, default 2^16

histogram_offset(*offset=0*)

Histogram time offset in ps.

(Documentation say ns, but it's most probably a typo.)

Parameters **offset** (*int*) – Histogram time offset in ps; 0, ... 500000

initialize(*mode='Histogram'*, *clock='Internal'*)
Initialize the device.

Parameters

- **devidx** (*int*) – index of the device
- **mode** (*string*) – operation mode, can be: 'Histogram'(default), 'T2','T3','Continuous'
- **clock** (*string*) – source of the clock, can be: 'External'(default), 'Internal'

input_CFD(*channel=0*, *level=50*, *zerox=0*)

UNUSED
Parameters of the input CFD (constant fraction divicriminator).
Values are given as a positive number although the electrical signals are actually negative.

Parameters

- **channel** (*int*) – input channel index; in our case 0 or 1
- **level** (*int*) – CFD discriminator level in millivolts
- **zerox** (*int*) – CFD zero cross level in millivolts

input_offset(*channel=0*, *offset=0*)
Input offset in time

Parameters

- **channel** (*int*) – input channel index; in our case 0 or 1
- **offset** (*int*) – time offset in ps -99999, ..., 99999

library_version
Version of the library.

load_config(*filename=None*)

Loads the yml configuration file of default instrument settings that probably nobody is going to change.
File are in folder /controller/picoquant/Hydraharp_controller.yml.

Parameters **filename** (*string*) – the name of the configuration file

number_input_channels
Number of installed input channels, in our case should be two (plus sync).

resolution
Resolution (in ps) at the current binning.

Return resolution resolution in ps at current binning

start_measurement(*acquisition_time=1000*)

Start acquisition.
Pay attention: acquisition_time is in ms!

Parameters **acquisition_time** (*int*) – Acquisition time in ms; 0.001, ... 360000

stop_measurement()

Stop acquisition.
Can be used before the acquisition time expires.

stop_overflow(*stop_at_overflow=0*, *stop_count=0*)

Determines if a measurement run will stop if any channel reaches the maximum set by stopcount.
In case of False, measurement will continue but counts above stop_count in any bin will be clipped.

Parameters

- **stop_at_overflow** (*bool*) – True is stop at overflow, False is do not stop (default True)
- **stop_count** (*int*) – Maximum counts at which the program stops because of overflow; 1, ... 4294967295 (default 4294967295)

sync_CFD(*level=50*, *zerox=0*)

UNUSED
Parameters of the sync CFD (constant fraction divicriminator).
Values are given as a positive number although the electrical signals are actually negative.

Parameters

- **level** (*int*) – CFD discriminator level in millivolts
- **zerox** (*int*) – CFD zero cross level in millivolts

sync_divider(*divider=1*)

Divider of the sync:
Must be used to keep the effective sync rate at values 12.5 MHz.

It should only be used with sync sources of stable period. Using a larger divider than strictly necessary does not do great harm but it may result in slightly larger timing

jitter. The readings obtained with HH_GetCountRate are internally corrected for the divider setting and deliver the external (undivided) rate. The sync divider should not be changed while a measurement is running.

Parameters

- **devidx** (*int*) – Index of the device (default 0)
- **divider** (*int*) – 1, 2, 4, 8 or 16

sync_offset(*value=0*)
Sync offset in time

Parameters **offset** (*int*) – time offset in ps -99999, ..., 99999

sync_rate()
Current sync rate

Return sync rate measured counts per second on the sync input channel

warnings

Warnings, bitwise encoded (see phdefin.h).
You must call HH_GetCoutRate and HH_GetCoutRate for all channels prior to this call.

Return warming warning message

warnings_text
Human readable warnings

Return warning warning in readable text

class hyperion.controller.picoquant.hydraharp.**Measurement_mode**
An enumeration.

class hyperion.controller.picoquant.hydraharp.**Reference_clock**
An enumeration.

hyperion.controller.picoquant.hydraharp.**c_int_p**
alias of hyperion.controller.picoquant.hydraharp.LP_c_int

LCC25 (thorlabs) driver

This controller supplies one class with several functions to communicate with the thorlabs LCC25 liquid crystal controller.

Note that this controller also implements units.

copyright 2020by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.thorlabs.lcc25.**Lcc**(*settings*)
This class is to controls the LCC25 thorlabs driver for a liquid crystal variable waveplate.

finalize()
Closes the connection to the device

freq
Modulation frequency when the operation mode is 'modulation' (mode = 0)

: getter : Asks for the current frequency

Returns The frequency value

Return type pint quantity

: setter :

Parameters **F** (*pint Quantity*) – frequency in Hz. It can be any value between 0.5 and 150Hz.

get_commands()
Gives a list of all the commands available :return: list with the commands available :rtype: str

get_voltage(*Ch=1*)
Gets the voltage value for VoltageCh where Ch = 1 or 2 for Voltage1 or Voltage2

Parameters **Ch** (*int*) – The channel to use, can be 1 or 2

Returns The method returns the voltage value present in the device in volts.

Return type pint quantity

idn()
Gets the identification for the device

Returns answer

Return type string

initialize()
Initialize the device

mode

Operation mode
The possible modes are:
1 = 'Voltage1' : sends a 2kHz sin wave with RMS value set by voltage 1
2 = 'Voltage2' : sends a 2kHz sin wave with RMS value set by voltage 2
0 = 'Modulation': sends a 2kHz sin wave modulated with a square wave where voltage 1 is one limit and voltage 2 is the second.
The modulation frequency can be changed with the command 'freq' in the 0.5-150 Hz range.

: getter : Gets the current mode : setter : Sets the mode

Parameters **mode** (*int*) – type of operation.

output
Tells if the output is enabled or not.

Getter

Returns output state

Return type logical

Setter

Parameters **state** (*logical*) – value for the amplitude to set in Volts

query(*message*)
Writes in the buffer and Reads the response.

Parameters **message** (*str*) – command to send to the device

Returns response from the device

Return type str

read()
Reads message from the device

Returns answer from the device (one line)

Type str

read_serial_buffer_in()

Reads everything the device has sent. By default it waits until a line is terminated by a termination character (

or

), but that check can be disabled using the input parameter.

param untill_at_least_one_termination_char defaults to True

type untill_at_least_one_termination_char bool

return complete serial buffer from the device

rtype bytes

set_voltage(*Ch*, *V*)
Sets the voltage value for Ch where Ch = 1 or 2 for Voltage1 or Voltage2

Parameters

- **Ch** (*int*) – channel to use (default =1)
- **V** (*pint quantity*) – voltage to set in Volts

write(*message*)
Sends the message to the device.

Parameters **message** (*string*) – the message to write into the device buffer

class hyperion.controller.thorlabs.lcc25.**LccDummy**(*settings*)
This is the dummy controller for the LCC25. The idea is to load this class instead of the real one to do testing of higher level functions without the need of the real device to be connected or working.

The logic is that this dummy device will respond as the real device would, with the correct type and size of information is expected.

This class inherits from the real device and the idea is to re-write only the init, the write and the read, so all the other functions remain the same and functioning.

The specific way to achieve this will be different for every device, so it has to be done separately.

To do so, we use a yaml file that tells the dummy class what are the properties of the device. For example, one property for the LCC25 is voltage1, which is the voltage for channel 1. Then from this you can build 2 commands: voltage1? to ask what is the value and voltage1=1 to set it to the value 1. So we build a command list using the CHAR ? and = for each of this properties.

Parameters

- **port** (*str*) – fake port name
- **dummy** (*logical*) – indicates the dummy mode. keept for compatibility

load_properties()
This method loads a yaml file with a dictionary with the available properties for the LCC25 and some defaults values. This dictionary is saved in properties and will be modified when a variable is writen, so the dummy device will respond with the previously set value.

query(*message*)
Writes in the buffer and Reads the response.

Parameters **message** (*str*) – command to send to the device

Returns response from the device

Return type str

read()
Dummy read. Reads the response buffer

write(*msg*)
Dummy write. It will compare the msg with the COMMANDS

Parameters **msg** (*str*) – Message to write

Osa controller

This is the controller for the optical spectrum analyzer (OSA), from Ando, model AQ6317B.

copyright 2020 by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.osa.osa_controller.**OsaController**(*settings*)
Class for OSA controller.

Parameters **settings** (*dict*) – this includes all the settings needed to connect to the device in question.

end_wav
" The end_wav is the end wavelength of the osa. :param end_wav: a value between 600 and 1750, must be greater than start_wav :type int

finalize()
This method closes the connection to the osa machine. This method should be called when all the things are done which you wanted to do with osa machine.

get_data()
Calculates the data created with the single sweep. Wait for OSA to finish before grabbing data

Return wav an list of the wavelengths, spec an list with spectrum data.

Rtype wav a list of floats and a list of floats

initialize()
Starts the connection to the device with given port

optical_resolution
" The optical resolution is the resolution of the spectrum you can take :param optical_resolution: a value that must be : 0.01, 0.02 ,0.05 ,0.1 ,0.2 ,0.5 ,1.0 ,2.0 , 5.0 :type float

perform_single_sweep()
Gives a command to the osa machine to perform a single sweep.

query(*msg*)
writes into the device message

Parameters **msg** (*string*) – command to write into the device port

Return ans answer from the osa

Rtype ans string

sample_points
" The amount of sample_points the osa machine must use in order to take a spectrum :param sample_points: the amount of points that will be on the x axis :type int

sensitivity
" The sensitivity of the osa machine :param sensitivity_string: a string that says how much the sensitivity must be :type string

set_settings_for_osa()
in this method the parameters for the osa machine are set with hand in order to quickly get results.

start_wav
" The start_wav is the start wavelength of the osa. :param start_wav: a value between 600 and 1750 :type int

wait_for_osa(*timeout=None*)
Method to let the program do nothing for a while in order to create enough time to let the osa machine take a spectrum.

Parameters **timeout** – time in seconds how long the program must wait before it resumes

if no timeout is specified a timeout will be calculated using self._time_constants :type timeout: float

class hyperion.controller.osa.osa_controller.**OsaControllerDummy**(*settings*)
In essence we have the same methods and we re-write the query to answer something meaningful but without connecting to the real device.

idn()
Identify command

Returns identification for the device

Return type string

initialize()
Dummy initialize

query(*msg*)
writes into the dummy device msg

Parameters **msg** (*string*) – command to write into the device port

read()
Fake read that returns always the value in the dictionary FAKE RESULTS.

Returns fake result

Return type string

write(*msg*)
Writes into the dummy device

Parameters **msg** (*string*) – message to be written in the device port

SK polarization driver

This class uses the 64bit dll from SK to use the SK polarization. For more details refer to the manual of the device.

For now it only supports one polarization analyzer connected.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.sk.sk_pol_ana.**Skpolarimeter**(*settings*)
This is the controller for the SK polarization. Based on their dll.

Parameters **settings** (*dict*) – dictionary with the entry 'dll_name' : 'SKPolarimeter' (default value)

finalize()
Closing communication with device

get_device_information()
Get SK polarization analyzer information. This function adds the obtained values to the properties of the class so they are accessible for all the functions

Returns reading answer from the function

Return type int

get_number_polarizers()
Get the number of polarizers available in the system

get_wavelength()
Get current wavelength

Returns wavelength in nm

Return type float

initialize(*wavelength=630*)
Initiate communication with the SK polarization analyzer

Parameters **wavelength** (*int*) – the working wavelength in nm

start_measurement()
start measurement

stop_measurement()
start measurement

wait_to_measure()
This function is meant to be used to delay the first measurement so the device is ready and producing data. It is used as a decorator.

class hyperion.controller.sk.sk_pol_ana.**SkpolarimeterDummy**(*settings*)
This is the dummy controller for the SK polarization. Based on their dll.

Stanford SR830

This controller (lock-in.py) supplies one class with several methods to communicate with the lock-in from "SRS" model SR830.

Based on the OSA Controller since it also uses GPIB connection.

copyright 2020by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.controller.stanford.sr830.**sr830**(*settings*)
The controller for the lock-in.

Parameters **settings** (*dict*) – this includes all the settings needed to connect to the device in question.

end_wav
" The end_wav is the end wavelength of the osa. :param end_wav: a value between 600 and 1750, must be greater than start_wav :type int

finalize()
This method closes the connection to the osa machine. This method should be called when all the things are done which you wanted to do with osa machine.

get_data()
Calculates the data created with the single sweep. Wait for OSA to finish before grabbing data

Return wav an list of the wavelengths, spec an list with spectrum data.

Rtype wav a list of floats and a list of floats

initialize()
Starts the connection to the device with given port

optical_resolution
" The optical resolution is the resolution of the spectrum you can take :param optical_resolution: a value that must be : 0.01, 0.02 ,0.05 ,0.1 ,0.2 ,0.5 ,1.0 ,2.0 , 5.0 :type float

perform_single_sweep()
Gives a command to the osa machine to perform a single sweep.

query(*msg*)
writes into the device message

Parameters **msg** (*string*) – command to write into the device port

Return ans answer from the osa

Rtype ans string

sample_points
" The amount of sample_points the osa machine must use in order to take a spectrum :param sample_points: the amount of points that will be on the x axis :type int

sensitivity
" The sensitivity of the osa machine :param sensitivity_string: a string that says how much the sensitivity must be :type string

set_settings_for_osa()
in this method the parameters for the osa machine are set with hand in order to quickly get results.

start_wav
" The start_wav is the start wavelength of the osa. :param start_wav: a value between 600 and 1750 :type int

wait_for_osa(*timeout=None*)
Method to let the program do nothing for a while in order to create enough time to let the osa machine take a spectrum.

Parameters **timeout** – time in seconds how long the program must wait before it resumes

if no timeout is specified a timeout will be calculated using self._time_constants :type timeout: float

Thorlabs motor controller

Note: this is a wrapper class around an external controller to fit it into hyperion structure

Is based on the imported Thorlabs APT python software from https://github.com/qpit/thorlabs_apt/tree/master/thorlabs_apt. This is installed in hyperion and can be found in C:/Users/NAME/AppData/Local/Continuum/anaconda3/envs/hyperion/Lib/site-packages/thorlabs_apt. This controller basically is just a wrapper to make things work within hyperion. The core is not always well documented, for better descriptions see the thorlabs_motor_instrument.

class hyperion.controller.thorlabs.tdc001_cube.**TDC001_cube**(*settings*)

This is the controller for the Thorlabs TDC001_cubes that work with motors.
It is just a wrapper that is linking to the thorlabs_apt.core with the class Motor in it.
Usually, the super().__init__() would execute the init of the BaseController.

However, since the current class inherits from both core.Motor and BaseController, only the first init is executed.

Parameters **settings** (*dict*) – the class Motor needs a serial number

finalize()

Here is should close the connection with the device, but this or similar functions do not exist in the core.Motor.
So this function is just so that higher layers dont give errors.

initialize()

The external core.Motor object is already initialized by executing super().__init__(self.serial).
So this function is just so that higher layers dont give errors.

class hyperion.controller.thorlabs.tdc001_cube.**TDC001_cubeDummy**(*settings*)
A dummy version that does not do anything at all.

RS 1316 driver

Very outdated code that does NOT follow hyperion structure and will not work.

copyright

(c)

license , see LICENSE for more details.

class hyperion.controller.rs.thermometer.**Rs1316**(*settings*)
controller class for the driver aa_mod18012 from AA optoelelectronics. This class has all the methods to communicate using serial.

NOTE: Our model has different ranges of frequency (see data sheet) Line 1 to 6: 82-151 MHz (this drives short wavelengths) Line 7 to 8: 68-82 MHz (this drives long wavelengths)

finalize()
This method closes the connection to the device. It is ran automatically if you use a with block

initialize()
Starts the connection to the device.

Experiment

In the Experiment folder we have a special set of classes the Experiments. This are classes talking to many instruments simultaneously and actually doing what is needed to measure some physical parameter of interest.

The basic example is to perform a scan: one physical parameter is sweep and for each of these values some other quantity is measured.

Every time we have to run an automatic experiment we write the class that contains all the methods needed to set it up, run and save the obtained data.

Of course, more complex experiments where many things are changed (or fixed) can be done.

Instruments

Here we group information about the second layer in our onion principle, the **Instrument**. We usually add functionalities that are specific to the setup or the experiment, not te devices itself, such as calibrations or specialized methods that are unlikely to be reused in another setup.

Base (Meta) Instrument

This file contains two base classes to be used as parent classes for Instrument classes and MetaInstrument classes.

class hyperion.instrument.base_instrument.**BaseInstrument**(*settings*)

Let Instrument classes inherit from this class. An instrument is a layer between the Controller (which takes care of hardware communication) and the user or Experiment. This instrument layer allows to add functionality, pint-units and may be used as an abstraction layer for similar controllers.

finalize()

This is to close connection to the device.

If you need to parse arguments to the finalize, make your own finalize method in your instrument class.

idn()

Identify command

initialize()

Starts the connection to the device.

If you need to parse arguments to the initialize, make your own initialize method in your instrument class.

load_controller(*settings*)

Loads controller

Parameters **settings** (*dict*) – dictionary with the field controller

Returns controller class

Return type class

class hyperion.instrument.base_instrument.**BaseMetaInstrument**(*settings*, *sub_instruments*)

Let MetaInstrument classes inherit from this class. A MetaInstrument takes other instruments as inputs (and does not use a controller)

The finalize() method of a MetaInstrument will call the finalize() methods of the sub instruments. In the case of an Experiment class, the MetaInstruments don't need to be finalized because the Instruments itself will be finalized.

Example Instrument

This is an example instrument, created to give developers a canvas to start their own instruments for real devices. This is only a dummy device.

class hyperion.instrument.example_instrument.**ExampleInstrument**(*settings*)

Example instrument. it is a fake instrument

amplitude

Gets the amplitude value :return: voltage amplitude value :rtype: pint quantity

finalize()

This is to close connection to the device. Note that the B

idn()
Identify command

Returns identification for the device

Return type string

initialize()
Starts the connection to the device"

AOTF instrument

This class (aa_aotf.py) is the model to connect to the AOTF using the controller aa_mod18012.py

The model is similar to the controller, but it adds specific functionalities such as units with Pint and some calibrations.

- **Wavelength calibration**: you can directly set the desired wavelength. For this it uses a look-up table and interpolation.

With this the class knows what voltages should be set when changing the wavelength.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.instrument.polarization.aa_aotf.**AaAotf**(*settings*)
This class is the instrument class for the AOTF driver.

It implements another layer in the MVC design, adding calibration and units functionality.

blanking(*state*, *mode='internal'*)
Define the blanking state. If True (False), all channels are on (off). It can be set to 'internal' or 'external', where external means that the modulation voltage of the channel will be used to define the channel output.

Parameters

- **state** (*logical*) – State of the blanking
- **mode** (*string*) – external or internal. external is used to follow TTL external modulation

choose_channel(*freq*)
This function selects an appropriated channel for a given frequency in the range supported by the device.

It returns channel 1 or 7 depending on the desired frequency.

- Channel 1 range: 82-151 MHz.
- Channel 7 range: 68-82 MHz.

Parameters **freq** (*pint Quantity*) – Frequency

Returns Channel number needed for the Frequency requested

Return type int

finalize()
To close the connection to the device

get_status()
Gets the status of all channels in the controller

load_calibration(*cal_file*)
This method loads the calibration file cal_file

Parameters **cal_file** (*string*) – calibration file complete path, including extension (txt expected)

set_all_values (*channel*, *freq*, *power*, *state=True*, *mode='internal'*)
Sets the values for freq, power, state and mode for channel. It can be used to turn in to or off, with state.

Parameters

- **channel** (*int*) – channel to use (can be from 1 to 8 inclusive)
- **freq** (*float*) – Frequency. Channels 1-6 supports(82,151)MHz and Channels 7 and 8, (68,82)MHz
- **power** (*float*) – Power to set in dBm (0 to 22)
- **state** (*logical*) – True for on and False for off
- **mode** (*string*) – 'internal' or 'external'

Returns Response from the driver

Return type string

set_defaults (*channel*)
Sets channel to default values given in the dictionary at the beginning of the class.

Parameters **channel** (*int*) – channel value to put to default settings.

set_frequency_all_range (*freq*, *power*, *state=True*, *mode='internal'*)
Automatically chooses channel 1 or 7 depending on the frequency requested and sets it.

Parameters

- **freq** (*pint Quantity*) – Frequency. Supported range (68,151) MHz
- **power** (*float*) – Power to set in dBm (0 to 22)
- **state** (*logical*) – True for on and False for off
- **mode** (*string*) – 'internal' or 'external'

set_wavelength (*wl*, *power*, *state=True*, *mode='internal'*)
This sets the wavelength wl by using the calibration file.

Parameters

- **wl** (*pint Quantity*) – This is the wavelength to set (it has to be in the range supported by the calibration file)
- **power** (*float*) – Power for the RF in the AOTF, in dBm. range = (0,22) dBm
- **state** (*logical*) – to turn the output on or off
- **mode** (*string*) – 'internal' or 'external'

Returns output from the driver.

Return type string

wavelength_to_frequency (*wl*, *method='interp'*)
uses the calibration file to calculate the frequency needed to get the wavelength wl.

Parameters

- **wl** (*pint Quantity*) – Wavelength (distance)
- **method** (*string*) – interpolation method ('interp' by default)

Returns Frequency value

Return type pint Quantity

Instrument for the function generator

This class (fun_gen.py) is the model to control the function generator agilent33522A It ads the use of units with pint.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.instrument.function_generator.fun_gen.**FunGen**(*settings*)
This class is to control the function generator.

Parameters **settings** (*dict*) – to parse the settings needed. Some keys are needed: 'controller' and 'instrument_id'

enable_output(*ch*, *state*)
Enables the output to the device

Parameters

- **ch** (*int*) – channel to use
- **state** (*logical*) – type to set the output on and off (True and False, respectively)

enable_voltage_limits(*channel*, *state*)
This function enables the limits for the maximum and minimum voltage output that can be generated by each channel. Those values are set with the method set_voltage_limits(channel, high, low).

This function enables this setting by putting state = true and disables it by putting state = false. When enable, setting a voltage outside the permitted values will give an error (not in python)

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **state** (*logical*) – True to turn on, False to turn of

finalize(*state=True*)
Closes the connection to the device

get_frequency(*channel*)
This functions gets the frequency output for the channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns frequency in the channel

Return type pint quantity

get_system_error()
This functions returns the error message

Returns error message

Return type string

get_voltage_high(*channel*)
Gets the high voltage value of the channel.

Parameters **channel** (*int*) – Channel number. Range depends on device

Returns current high voltage for the channel

Return type pint quantity

get_voltage_limits(*channel*)
Gets the set values for the voltage limits, high and low.

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns array with [high_value, low_value]

Return type pint quantity array

get_voltage_limits_state(*channel*)
This function gets the state of the voltage limits for the required channel

Parameters **channel** (*int*) – number of channel. it can be 1 or 2 for this model

Returns current state of the channel voltage limits

Return type logical

get_voltage_low(*channel*)
Gets the low voltage value of the channel.

Parameters **channel** (*int*) – Channel number. Range depends on device

Returns current low voltage for the channel

Return type pint quantity

get_voltage_offset(*channel*)
Gets the peak-to-peak voltage of the channel.

Parameters **channel** (*int*) – Channel number. Range depends on device

Returns current offset for the channel

Return type pint quantity

get_voltage_vpp(*channel*)
Gets the peak-to-peak voltage of the channel.

Parameters **channel** (*int*) – Channel number. Range depends on device

Returns current peak to peak voltage

Return type pint quantity

get_waveform(*channel*)
Sets the waveform to be generated. The functions available are FUNCTIONS = ['SIN','SQU','TRI','RAMP','PULS','PRBS','NOIS','ARB','DC']

Parameters **channel** (*int*) – Channel number

Returns One of the FUNCTIONS

idn()
Ask for the identification

Returns message with identification from the device

Return type string

load_defaults(*apply*, *filename=None*)

This loads the default configuration and applies it, depending on the apply parameter. The information is kept in the variable self.DEFAULTS

Parameters

- **apply** (*logical*) – decides weather to apply this settings or not

- **filename** – location for the config_agilent33522A.yml to use. If not given uses

the default config file in the model/function_generator folder

output_state()
Gets the current state of the output

Returns A list of with the state of all channels.

Return type list of logical

set_frequency(*channel*, *freq*)
This functions sets the frequency output for the channel

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **freq** (*pint quantity*) – Frequency to set

set_voltage_high(*channel*, *value*)
Sets the high voltage value for the channel.

Parameters

- **channel** (*int*) – Channel number. (list in self.CHANNELS)
- **value** (*pint quantity*) – The input value in Volts

Returns current high voltage for the channel

Return type pint quantity

set_voltage_limits(*channel*, *high*, *low*)
This function generator can set a minimum and maximum voltage value that will not be exceeded to protect the device that is being feed. This function sets the values high and low as voltage limits. If this option is activated then the output will not exceed the given values. NOTE: setting the values does not activate this feature. To enable/disable it use the function

Parameters

- **channel** (*int*) – number of channel. it can be 1 or 2 for this model
- **high** (*pint quantity*) – upper voltage limit
- **low** (*pint quantity*) – lower voltage limit

set_voltage_low(*channel*, *value*)
Sets the low voltage value for the channel.

Parameters

- **channel** (*int*) – Channel number. (list in self.CHANNELS)
- **value** (*pint quantity*) – voltage low to set to channel

Returns voltage set

Return type pint quantity

set_voltage_offset(*channel*, *value*)
Sets the DC offset voltage of the channel.

Parameters

- **channel** (*int*) – Channel number. Range depends on device
- **value** – The input value in Volts

:type pint quantity :return: current offset for the channel :rtype: pint quantity

set_voltage_vpp(*channel*, *value*)
Sets the peak-to-peak voltage of the channel.

Parameters

- **channel** (*int*) – Port number. Range depends on device
- **value** (*pint quantity*) – peak to peak voltage to set

Returns current peak to peak voltage

Return type pint quantity

set_waveform(*channel*, *fun*)
Sets the waveform to be generated. The functions available are FUNCTIONS = ['SIN','SQU','TRI','RAMP','PULS','PRBS','NOIS','ARB','DC']

Parameters

- **channel** (*int*) – channel to use
- **fun** (*string*) – function to generate

SK Polarimeter

This class (polarization.py) is the model to connect to the SK Polarization analyzer from SK.

The model is similar to the controller, but it adds specific functionalities such as units with Pint and error descriptions

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.instrument.polarization.polarimeter.**Polarimeter**(*settings*)
This class is the model for the SK polarization.

change_wavelength(*w*)
Change the current wavelegnth to w

Parameters **w** (*pint quantity (distance)*) – wavelength

Returns current wavelength

Return type pint quantity (distance)

create_header(*errors=False*, *extra_name=None*, *extra_unit=None*, *extra_description=None*)
creates the header to save the data

Parameters

- **extra_name** (*str*) – if a first column has to be added, this is the name
- **extra_unit** (*str*) – if a first column has to be added, this is the unit
- **extra_description** (*str*) – if a first column has to be added, this is the description

Returns header with the information of the data saved

Return type string

finalize()
Finishes the connection to the SK polarization

get_average_data(*N*)
Takes N data points and gives back the average and the error (std)

Returns av, array with dim 13 containing the average of the N measurements for each property in DATA_TYPES, and the std

Return type numpy array

Returns st, array with dim 13 containing the std of the N measurements for each property in DATA_TYPES, and the std

Return type numpy array

Example:

```
>>> av, st = get_average_data(10)
>>> print('Values: {} +/- {}'.format(av, st))
Values
```

get_data()
This methods gets the a single measurement point from the device. It reads all the types of data available, listed in DATA_TYPES.

Returns a list with the data

Return type list

get_information()
gets the information from the device: number of polarizers and id.

get_multiple_data(*N*)
This is to get a stream of data directly from the device.

Parameters **N** (*int*) – number of data points needed

Returns a np array

get_wavelength()
asks the device the current wavelength.

Returns current wavelength

Return type pint quantity

initialize(*wavelength=None*)
This is to initialize the model by calling the initizialize function in the controller. It adds units for the wavelength

Parameters **wavelength** (*pint quantity*) – the working wavelength

save_as_netCDF4(*filename*, *data*)
Saves the data from the polarimeter measurement into a netCDF4 file.

save_data(*data, extra=[None, None, None, None], file_path='polarimeter_test.txt'*)
saves the data. It assumes that the data comes from the SK so it is a matrix of 13 rows and an arbitrary number of columns.

Parameters

- **data** (*numpy array*) – array of (N,13)
- **extra** – list containing the extra vector to save (1xN), the name of the extra data and

the unit. For example: [np.array([1,2,3]),'time','second','elapsed time']. :type extra: list

start_measurement()
This method starts the measurement for the polarization analyzer.

stop_measurement()
This method stops the measurement for the polarization analyzer.

LCC25 (thorlabs) model

This class (variable_waveplate_gui.py) is the model to drive the LCC25 variable waveplate controller.

It ads the use of units with pint and the wavelength calibration to obtain make the variable waveplate a quarter waveplate for a given wavelength.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.instrument.polarization.variable_waveplate.**VariableWaveplate**(*settings*)
This class is the model for the LCC25 analog voltage generator for the variable waveplate from thorlabs.

do_interp(*w*, *x*, *y*)
This function interpolates the voltage value for the wavelength value w, using the calibration data (x,y) where x is wavelength and y is voltage

Parameters

- **w** (*pint quantity*) – desired wavelength
- **x** (*pint quantity*) – wavelength vector from calibration
- **y** (*pint quantity*) – calibrated voltage values

Returns the voltage for the desired wavelength

Return type pint quantity

finalize(*state=False*)
Closes the connection to the device

freq
Modulation frequency when the operation mode is 'modulation' (mode = 0)

: getter :

Asks for the current frequency

Returns The frequency value

Return type pint quantity

: setter :

Parameters **F** (*pint Quantity*) – frequency in Hz. It can be any value between 0.5 and 150Hz.

get_analog_value(*channel*)
Gets the analog voltage of the channel.

Parameters **channel** (*int*) – Port number. Range depends on device

Returns voltage: voltage in use for the channel

Return type pint quantity

idn()
Ask for the identification

initialize()
initializes the connection with the controller

load_calibration(*cal_file*)
This method loads the calibration file cal_file

Parameters **cal_file** (*string*) – calibration file complete path, including extension (txt expected)

mode
Operation mode

The possible modes are:

'Modulation': sends a 2kHz sin wave modulated with a square wave where voltage 1 is one limit and voltage 2 is the second. the modulation frequency can be changed with the command 'freq' in the 0.5-150 Hz range.

'Voltage1' : sends a 2kHz sin wave with RMS value set by voltage 1

'Voltage2' : sends a 2kHz sin wave with RMS value set by voltage 2

'QWP' : uses the calibration file to set the voltage that acts as a quarter-wave plate for the specified wavelength.

: getter :

Gets the current mode

Returns current mode

Return type string

: setter :

Sets the mode

Parameters **mode** (*string*) – type of operation.

output
Tells if the output is enabled or not.

Getter

Returns output state

Return type logical

Setter

Parameters **state** (*logical*) – value for the amplitude to set in Volts

quarter_waveplate_voltage(*wavelength*, *method='lookup'*)
This method gives the voltage needed to set on the LCC25 to get a quarter waveplate (QWP) behaviour for a given wavelength. It is based on a calibration file that relates the QWP voltage to a wavelength and (so far) uses a linear fit to those data points.

Parameters

- **wavelength** (*pint Quantity*) – The input wavelength
- **method** (*string*) – method to extrapolate between measured data points

Returns the QWP voltage

Return type pint quantity

set_analog_value(*channel*, *value*)
Sets the analog voltage of the channel.

Parameters

- **channel** (*int*) – Port number. Range depends on device
- **value** (*pint quantity*) – The input value in Volts between 0 and 25 (Pint type)

set_quarter_waveplate_voltage(*wavelength*)
This method sets the quarter-wave plate (QWP) voltage to the variable-wave plate for a given wavelength. It uses Voltage 1 for output.

Parameters **wavelength** (*pint Quantity*) – The input wavelength

Returns the voltage set to the controller

Return type pint quantity

Hydraharp Instrument

This is the instrument level of the correlator Hydraharp400 from Picoquant

class hyperion.instrument.correlator.hydraharp_instrument.**HydraInstrument**(*settings*)
A class for the Hydraharp instrument.

Parameters **settings** (*dict*) – to parse the needed settings.

configurate(*filename=None*)

Loads the yml configuration file of default instrument settings that probably nobody is going to change. File in folder instrumentcorrelatorHydrahaInstrument_config.yml.

Parameters **filename** (*string*) – the name of the configuration file

count_rate(*channel*)
Asks the controller the rate of counts on the count channels and adds units.

Parameters **channel** (*int*) – count rate channel 0 or 1 (1 or 2 on device) connected to the photon counter

Returns count rate that is read out in counts per second

Return type pint quantity

finalize()
This method is to close connection to the device.

initialize()
Starts the connection to the device, calibrates it and configures based on the yml file.

make_histogram(*integration_time*, *count_channel*)

Does the histogram measurement, checking for the status, saving the histogram.
You need to start the measurement and than you could collect the histogram.
This communicates with the controller method start_measurement and than goes to the wait_till_finished method.

Parameters

- **count_channel** (*int*) – number of channel that is correlated with the sync channel, 1 or 2
- **integration_time** (*pint quantity*) – acquisition time of the histogram; **(please don't use the word time)**

Returns array containing the histogram

Return type array

set_histogram(*leng*, *res*)

Clears the possible previous histogram, sets the histogram length and resolution.
Has also to do with the binning and the length of the histogram.
In the correlator software, the length is fixed to 2^16 and the resolution determines the binning and thus the time axis that is plot.

Parameters

- **leng** (*int*) – length of histogram
- **res** (*pint quantity*) – resolution in the histogram in ps

stop_histogram()

This method stops taking the histogram, could be used in higher levels with a thread.

sync_rate()
Asks the controller the rate of counts on the sync channel and adds units.

Returns counts per second on the sync channel

Return type pint quantity

wait_till_finished(*integration_time*, *count_channel*)

This method should ask the device its status and keep asking until it's finished.
However, the remaining time is printed but not shown with the timer in the gui.
The loop breaks if self.stop is put to True, which could be done in a higher level with a thread.

Parameters

- **integration_time** (*pint quantity*) – integration time of histogram **(please don't use the word time)**
- **count_channel** (*int*) – number of channel that is correlated with the sync channel, 1 or 2

Returns remaining time in seconds

Return type pint quantity

Osa Instrument

This is the osa instrument, created to be able to indirectly talk to the device through controller and get data which can be shown in the gui. So request for doing things go view > instrument > controller

This class uses pint to give units to the variables.

class hyperion.instrument.spectrum.osa_instrument.**OsaInstrument**(*settings*)

OsaInstrument class

finalize()

Closes the connection to the osa machine

idn()

Identify command

Returns identification for the device

Return type string

initialize()

Starts the connection to the osa machine

is_end_wav_bigger_than_start_wav(*end_wav*, *start_wav*)

Check to see if end_wav is bigger than the start_wav

Parameters

- **end_wav** – a pint quantity
- **start_wav** – a pint quantity

Type pint nm quantity

Type pint nm quantity

Returns true if condition passed, false if condition failed.

:rtype boolean

is_end_wav_value_correct(*end_wav*)

Is end_wav in range between 600 and 1750

Parameters **end_wav** – a pint quantity

Type pint nm quantity

Returns true if condition passed, false if condition failed.

:rtype boolean

is_start_wav_value_correct(*start_wav*)

Is start_wav in range between 600 and 1750 nm

Parameters **start_wav** – the startwavelength

Type pint nm quantity

Returns true if condition passed, false if condition failed.

:rtype boolean

load_config(*filename=None*)

Loads the configuration file to generate the properties of the Scan and Monitor.

Parameters **filename** – Path to the filename. Default is: 'Config/experiment.yml' if not specified.

:type string

take_spectrum()
Method where a spectrum will be taken using the osa machine.

Returns wav, spec: two list containing the data from the taken spectrum.

Rtype wav, sepec wav(a list of floats), spec(a list of floats)

ANC350 Attocube Instrument

This is the instrument level of the position ANC350 from Attocube (in the Montana)

class hyperion.instrument.position.anc_instrument.**Anc350Instrument**(*settings*)
Anc 350 instrument class.

capacitance(*axis*)
Measures the capacitance of the stepper or scanner; no idea why you would want to do that.

Parameters **axis** (*string*) – scanner axis to be set, XPiezoScanner, YPiezoScanner, XPiezoStepper, etc.

check_if_moving(*axis*, *position*)

work in progress!
Checks whether the piezo is actually moving.
It checks if you are not out of range, or putting a too low voltage.
If that's okay, it keeps checking whether you are actually moving.
However, the status of the piezo is not always correct, and the movement is not linear, so this method is not finished yet.
It also keeps checking whether self.stop is True, and asking the position. This can be used in higher levels with threads and timers.

Parameters

- **axis** (*pint quantity*) – scanner axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **position** – absolute position that you want to go to; needs to be an integer, no float!

Returns The end position, so the moving method can compare that with the start position

configure_scanner(*axis*)

- Does the necessary configuration of the Scanner:
- you need to set the mode to INT, not DC-IN

Parameters **axis** (*string*) – scanner axis to be set, XPiezoScanner, YPiezoScanner or ZPiezoScanner

configure_stepper(*axis*, *amplitude*, *frequency*, *amplitudeControl*)

- Does the necessary configuration of the Stepper:
- the Stepper seem to perform best when put in Amplitude Control mode, nr. 1; you can change that however
- loads the actor file, files are in controller folder, their names hardcoded in controller init

- sets the amplitude and frequency
- the amplitude influences the step width, the frequency influences the speed
- also stores the current position of the axis in self.current_positions

Parameters

- **axis** (*pint quantity*) – stepper axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **amplitude** – amplitude voltage; at room temperature you need 30V-40V, at low temperatures 40V-50V, max 60V; high amplitude is large stepwidth
- **frequency** – frequency to be set; higher means more noise but faster; between 1Hz and 2kHz

finalize()
This is to close connection to the device.

get_position(*axis*)
Asks the position from the controller level. This method is useful in higher levels where you want to display the position.

Parameters **axis** (*string*) – stepper axis, XPiezoStepper, YPiezoStepper, or XPiezoScanner, etc.

given_step(*axis*, *direction*, *amount*)

Moves by a number of steps that theoretically should be determined by the set amplitude and frequency; in practice it's different.
You have to give it a lot of time, things break if you ask too much whether it is finished yet.

Parameters

- **axis** (*string*) – stepper axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **direction** (*integer*) – direction to move: forward = 0, backward = 1
- **amount** (*integer*) – amount of steps that you want to take

initialize()

Starts the connection to the device by initializing the controller.
Loads the axis names from the yml file.
Runs set_temperature_limits.

move_continuous(*axis*, *direction*)
Keeps moving the stepper axis until you manage to stop it (for which you need threading).

Parameters

- **axis** (*string*) – stepper axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **direction** (*integer*) – direction to move: forward = 0, backward = 1

move_relative(*axis*, *step*)

Moves the Stepper by an amount to be given by the user.
Pay attention: does not indicate if you take a position outside of the boundary, but you will keep hearing the noise of the piezo.

Parameters

- **axis** (*string*) – stepper axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **step** (*pint quantity*) – amount to move, can be both positive and negative; needs to be an integer, no float!

move_scanner(*axis*, *voltage*)

Moves the Scanner by applying a certain voltage.
Pay attention: the maximum voltage depends on the temperature in the cryostat.
There is no calibration, so you don't know how far; but the range is specified for 50um with a voltage of 0-140V.

Parameters

- **axis** (*string*) – scanner axis to be set, XPiezoScanner, YPiezoScanner or ZPiezoScanner
- **voltage** (*pint quantity*) – voltage to move the scanner; from 0-140V

move_to(*axis*, *position*)

Moves to an absolute position with the Stepper and tells when it arrived.
Pay attention: does not indicate if you take a position outside of the boundary, but you will keep hearing the noise of the piezo.

Parameters

- **axis** (*pint quantity*) – stepper axis to be set, XPiezoStepper, YPiezoStepper or ZPiezoStepper
- **position** – absolute position that you want to go to; needs to be an integer, no float!

set_temperature_limits()
The maximum voltage to put on the piezo scanners depends on the temperature in the cryostat. The user has to give that. The maximum ranges from 60V at room temperature to 140V at 4K, and everything in between is linearly interpolated.

stop_moving(*axis*)
Stops moving to target/relative/reference position.

Parameters **axis** (*string*) – scanner or stepper axis to be set, XPiezoStepper, XPiezoScanner, YPiezoScanner etc

update_all_positions()
Uses self.get_position to ask the position on all axes. This method is useful in higher levels where you want to display the position.

zero_scanners()
Puts 0V on all three Piezo Scanners.

Thorlabs thorlabs_motor Instrument

Connects to the tdc001_cube controller, which is just a wrapper for the core underneath that we installed. You can find the core here https://github.com/qpit/thorlabs_apt/tree/master/thorlabs_apt, or need to copy it into C:/Users/NAME/AppData/Local/Continuum/anaconda3/envs/hyperion/Lib/site-packages/thorlabs_apt.

I implemented and documented those functions that I am actually going to use. If you want to use others, they might exist in the core.

class hyperion.instrument.position.thorlabs_motor_instr.**Thorlabsmotor**(*settings*)

Thorlabsmotor instrument

check_move(*value*)

First checks whether there is no limit reached.

Then checks whether the units actually agree with the kind of device, so degrees for a waveplate or a length for a stage motor.

Returns whether you can move, otherwise prints warnings.

Returns the units of the value, since the controller thinks in degrees and mm.

Parameters **value** (*pint quantity*) – distance or new position that you would want to move

Returns you_can_move, value

Return type Bool, float

finalize()

This would have been to close connection to the device, but that method does not exist in the core controller.

get_axis_info()

Important returns axis information of stage.

If units = 1, the units are in mm and the device is a stage motor.

If units = 2, the units are in degrees and the device is a motorized waveplate.

Store this in self.kind_of_device, so the rest of this class knows.

Returns (minimum position, maximum position, stage units, pitch)

Return type tuple

initialize()

- Initializes the cube:
- checks whether the serial number is recognized by the cube
- runs list_device to check whether this serial number actually exists in all connected T-cubes
- asks the harware info just in case
- blinks the light to identify the T-cube
- runs set_axis_info, which will set the minimum position to -12 and
- run get_axis_info, which will figure out whether you are connected to a waveplate or stage motor **always run this one!!**

is_in_motion()
Returns whether thorlabs motor is in motion, and prints a warning if so.

Returns in motion

Return type bool

list_devices()

Lists all available devices.
It actually maybe should live outside of this class, since it talks to all Thorlabs motors attached, not a single one.
However, I could not make that work, so now I kept it here.
It also runs through the list now and checks whether the serial of the T-cube that you want to talk to, exists in the list.

make_step(*stepsize*, *blocking*)

Moves the T-cube by one step of a stepsize.
Actually just uses move_relative, but I thought maybe this method might be useful for a gui.
If blocking is True, it will move until its done.
If blocking is False, it might not reach its destination if you dont give it time.

Parameters

- **stepsize** (*pint quantity*) – stepsize in mm or degree
- **blocking** (*bool*) – wait until moving is finished; default False

motion_error()
Returns whether there is a motion error (= excessing position error), and prints a warning if so.

Returns motion error

Return type bool

motor_current_limit_reached()
Return whether current limit of thorlabs_motor has been reached, and prints a warning if so.

Returns current limit reached

Return type bool

move_absolute(*new_position*, *blocking*)

Moves the T-cube to a new position, but first checks the units by calling check_move.
The method check_move will give back the correct units.
If blocking is True, it will move until its done.
If blocking is False, it might not reach its destination if you dont give it time.

Parameters

- **new_position** (*pint quantity*) – the new position
- **blocking** (*bool*) –

move_home(*blocking*)

Moves to home position.
You can use the blocking method of the core, but than higher layers will not be able to stop the move or know the position.
So I implemented my own blocking that uses the is_in_motion method.
If blocking is True, it will move until its done.
If blocking is False, it might not reach its destination if you dont give it time.

Parameters **blocking** (*bool*) – wait until homed

move_relative(*distance*, *blocking*)

Moves the T-cube with a distance relative to the current one, but first checks the units by calling check_move.
The method check_move will give back the correct units.
If blocking is True, it will move until its done.
If blocking is False, it might not reach its destination if you dont give it time.

Parameters

- **value** (*pint quantity*) – relative distance in mm or degree
- **blocking** (*bool*) – wait until moving is finished; default False

move_velocity(*direction*, *blocking*)

Moves the T-cube with a certain velocity until it gets stoped.
The method check_move will give back the correct units.
If blocking is True, it will move until its done.
If blocking is False, it might not reach its destination if you dont give it time.

Parameters

- **blocking** (*bool*) – wait until moving is finished; default False
- **direction** (*1 for forward 2 for backward*) – direction of movement

moving_loop()

This method is used by the move_home, move_relative and move_absolute methods.
It stays in the while loop until the position is reached or self.stop is set to True,
meanwhile updating the current_position as known in this instrument level.
This means it can be used in higher levels to display the position on the gui and thread the stop function.
If the sleeps are making your program too slow, they could be changed.
Pay attention not to make the first sleep too short, otherwise it already starts asking before the guy even knows whether he moves.

position()

Asks the position to the controller and returns that.
Units depend on the kind of device; either mm or degrees.
Remembers the current_position as declared in the init.

Returns position in mm or degrees

Return type pint quantity

set_axis_info()

Executes get_axis_info to set the kind of device, and changes the minimum position to -12.0.
This is important because the stage axis puts itself at 0 if you shut down the cubes,
so it happens that afterwards you want to go to a negative position.
To prevent errors, this method changes the minimum position from 0 to -12.0.

stop_moving()

Stop motor but turn down velocity slowly (profiled).

Winspec Instrument

Aron Opheij, TU Delft 2019

IMPORTANT REMARK: In the current implementation it is not possible to use this instrument in threads.

Tips for finding new functionality:

Once you have an WinspecInstr object named ws, try the following things: This will list all keywords: [key for key in ws.controller.params] There are shorter lists with only experiment (EXP) and spectrograph (SPT) commands: [key for key in ws.controller.params_exp] # note that prefix **EXP_** is removed [key for key in ws.controller.params_spt] # note that prefix **SPT_** is removed To filter in those you could try: [key for key in ws.controller.params_exp if 'EXPOSURE' in key] [key for key in ws.controller.params_spt if 'GROOVES' in key]

To request the value for a keyword try: ws.controller.exp_get('EXPOSURETIME') ws.controller.exp_get('GRAT_GROOVES')

See also:

<no title>

class hyperion.instrument.spectrum.winspec_instr.**WinspecInstr**(*settings*)

Winspec Instrument

Parameters **settings** (*dict*) – this includes all the settings needed to connect to the device in question.

Information for settings dict: To overwrite certain possible values add a key conaining a list of possible values to the settings dict: For example, if your WinSpec does not have 'Normal' mod in shutter_controls, add this key: shutter_controls: - Closed - Opened

accumulations

attribute: Number of Accumulations.

Getter Returns the Number of Accumulations set in Winspec

Setter Attempts to updates the Number of Accumulations in Winspec if required. Gives warning if failed.

Type int

ascii_output

attribute: Also save as ASCII file (next to SPE)

Getter Returns if ASCII save is enabled.

Setter Sets saving as ASCII file.

Type bool

autosave
attribute: Auto-save and prompts: default possible values: 'Ask', 'Auto', 'No'

Type str

avalanche_gain
Get the avalanche gain (only available with EM CCD camera)

Getter Returns avalanche gain in Winspec

Setter Tries to change avalanche gain in Winspec if required.

Type int

bg_file
attribute: Full path to Background File Note: it does not check if the file exists or is it is a valid file.

Type string

bg_subtract
attribute: Background Subtraction

Getter Returns if Background Subtraction is enabled.

Setter Sets Background Subtraction.

Type bool

ccd
attribute: CCD Readout mode. default possible values: 'Full' 'ROI'

Type str

central_nm
attribute: Central position of grating in nm

Getter Returns current central position of grating

Setter Rotates grating to new nm position. Waits for it to complete.

Type float

collect_spectrum(*wait=True*, *sleeptime=True*)

Retrieves the last acquired spectrum from Winspec.
There are a few possibilities:
- you are not using the autosave of winspec: there will be no sleeps
- you are using the autosave of winspec, but not saving the ascii: there will be no sleeps
- you are using the autosave of winspec, and saving ascii: you need some sleeptime, otherwise things break
- in case of acquiring a spectrum, 0.1s is enough for sleeping
- in case of acquiring an image, you need 1s in between and 2s afterwards
Winspec does wait while acquiring data, but does not send a wait command when it is autosaving.
So the combination of autosaving big ascii data in a scan would cause problems without the sleep.
Pay attention: if you do get the sleeps unwanted, check whether you are sending the correct config yml file!

Parameters

- **wait** (*bool*) – If wait is True (DEFAULT) it will wait for WinSpec to finish collecting data.
- **sleeptime** (*bool*) – Sleeptime adds some sleeps to make sure Winspec has enough time to Autosave ascii images

Returns list or nested list

confirm_overwrite
attribute: Confirm Overwrite File

Type bool

current_temp
read-only attribute: Temperature measured by Winspec in degrees Celcius.

Getter Returns the Temperature measured by Winspec.

Type float

delay_time_s
attribute: The delay time under Experiment/Timing in seconds

Type float

display_flip
attribute: Display Flip (up-down).

Getter Returns if Display Flip is set in Winspec.

Setter Sets Display Flip in Winspec.

Type bool

display_reverse
attribute: Display Reverse (left-right)

Getter Returns if Display Reverse is set in Winspec.

Setter Sets Display Reverse in Winspec.

Type bool

display_rotate
attribute: Display Rotate.

Getter Returns if Display Rotation is set in Winspec.

Setter Sets Display Rotation in Winspec.

Type bool

exposure_time
attribute: Exposure Time.

Getter Returns the Exposure Time set in Winspec

Setter Attempts to updates the Exposure Time in Winspec if required. Gives warning if failed.

Type Pint Quantity of unit time

fast_safe
attribute: Fast or Safe mode in Experiment/Timing default possible values: 'Fast', 'Safe'

Type str

file_increment
auto increments file or not

Type bool

filename
Note that this needs to be set before acquiring a spectrum. If you want to store the SPE files as well, the best approach is to make sure self.autosave = 'Auto'

use take_spectrum('new_name.SPE') :return:

finalize()
Mandatory function. Get's called when exiting a 'with' statement.

gain
attribute: ADC Gain value.

Getter Returns Gain set in Winspec

Setter Attempts to updates Gain setting in Winspec if required. Gives warning if failed.

Type int

getROI()
Retrieve current Region Of Interest. :return: list containing [top, bottom, v_binsize, left, right, h_binsize]

grating
attribute: Grating number (starts at 1)

Getter Returns current grating number

Setter Switched to new grating. Waits for it to complete.

Type int

idn()
Identify command

Returns Identification string of the device.

Return type string

initialize()
Starts the connection to the Winspec softare and retrieves parameters.

is_acquiring
Read only property that indicates if WinSpec is still busy acquiring. Returns True or False.

nm_axis()
Returns list with nm axis values of last collected spectrum.

saveas(*filename*)
Make WinSpec save current spectrum to disk (in format specified in WinSpec). Note: I's also possible to use autosave function of WinSpec: self.autosave = 'Auto'

Parameters **filename** (*string*) – The full path to save the file. If None specified it used default name.

Returns

setROI(*top='full_im'*, *bottom=None*, *v_binsize=None*, *left=1*, *right=None*, *h_binsize=1*)
Note for the new camera (the 1024x1024 one) the horizontal range needs to be a multiple of 4 pixels. If the users input fails this criterium, this method will expand the range. Also the v_group and h_group, need to fit in the specified range. If the input fails, a suitable value will be used. And the user will be warned.

Parameters

- **top** – Top-pixel number (inclusive) (integer starting at 1). Alternatively 'full_im' (=DEFAULT) of 'full_spec' can be use.
- **bottom** – Bottom-pixel number (integer). DEFAULT value is bottom of chip
- **v_binsize** – Vertical bin-size in number of pixels (integer). None sums from 'top' to 'bottom'. DEFAULT: None
- **left** – Left-pixel number (inclusive) (integer starting from 1). DEFAULT is 1
- **right** – Right-pixel number (integer). DEFAULT is rightmost pixel
- **h_binsize** – Horizontal binning (integer), DEFAULT is 1

Returns

Examples

```
>>>
setROI('full_im')     returns the full CCD
setROI('full_spec')   returns the full CCD, summed vertically to␣
↪result in 1D array
setROI(51)            sums from pixel 51 to the bottom
setROI(51, 70)        sums vertically from pixel 51 to 70
setROI(51, 70, 20)    sums vertically from pixel 51 to 70
setROI(41, 60, 5)     result in 4 bins of 5 pixles
setROI(41, 60, 1)     no binning, result will be 20 pixels high
setROI(41, 60, None, 101, 601)      modify horizontal range
setROI(41, 60, None, 101, 601, 10)  apply horizontal binning of 10␣
↪pixels (result will be 50 datapoints wide)
```

shutter_control
attribute: Shutter Control default possible values: ? , 'Free Run', ? , 'External Sync'

Type str

spec_mode
attribute: True for Spectroscopy Mode, False for Imaging Mode

Type bool

start_acquiring(*name=None*)
Starts acquisition of spectrum. Does not wait for it to finish. If name is specified. The last spectrum in WinSpec is closed and a new spectrum with the specified name is created. If name is None (DEFAULT) the current spectrum will be overwritten.

Parameters **name** (*string*) – Full path to file to store. Or None, for using default.

take_spectrum(*name=None*, *sleeptime=True*)
Acquire spectrum, wait for data and collect it. Performs start_acquiring(name), followed by collect_spectrum(True). See those methods for more details.

take_spectrum_alt(*name=None*, *sleeptime=True*)
Acquire spectrum, wait for data and collect it. Performs start_acquiring(name), followed by collect_spectrum(True). See those methods for more details.

target_temp
attribute: Detector target temperature in degrees Celcius.

Getter Returns Target Temperature set in Winspec

Setter Attempts to updates Target Temperature in Winspec if required. Gives warning if failed.

Type float

temp_locked
read-only attribute: Temperature locked state measured by Winspec in degrees Celcius.

Getter Returns True is the Temperature is "locked"

Type bool

timing_mode
attribute: Timing Mode default possible values: 'Normal', 'Closed', 'Opened'

Type str

Beam Flags Instrument

Instrument for homebuilt Arduino based beam flags. Designed to work with Arduino running:

qnd_simple_double_flag_controller.ino "QND Simple Double Flag Controller, version 0.1, date 2019-09-17"

class hyperion.instrument.misc.beam_flags_instr.**BeamFlagsInstr**(*settings*)
Beam Flags Instrument Intended to be used with an arduino running:

qnd_simple_double_flag_controller.ino "QND Simple Double Flag Controller, version 0.1, date 2019-09-17"

Parameters **settings** (*dict*) – This includes all the settings needed to connect to the device in question.

f1
bool: Set/get flag with label '1' (True for 'g', False for 'r')

f2
bool: Set/get flag with label '2' (True for 'g', False for 'r')

f3
bool: Set/get flag with label '2' (True for 'g', False for 'r')

finalize()
Closes the connection to the device.

get_flag(*flag_number*)
Get flag state as bool. Identify flag by its number (in stead of string)

Parameters **flag_number** (*int*) – The number indicating the flag

Returns True for 'g' and False for 'r', (None for other)

Return type bool

get_specific_flag_state(*flag_name*)
Query the state of a specific flag. Also updates this Instruments internal state if the flag name occurs in the dictionary. (Does not check for valid flag_name)

Parameters **flag_name** (*string*) – the name of the flag

Returns state of the flag

Return type string

idn()
Identify command.

Returns Identification string of the device (if it has it)

Return type str

initialize()
Starts the connection to the device.

passive_update_from_manual_changes()
When toggle switches are manually changed, the arduino will send messages like 1g or 2r. This method will read the Serial buffer-in and update this Instruments internal state of the flags according to the last states found in the buffer.

Returns if it changed any state

Return type bool

set_flag(*flag_number*, *bool_state*)
Set flag using its number and bool state.

Parameters

- **flag_number** (*int*) –
- **bool_state** (*bool*) – flag state (True for 'g', False for 'r')

set_specific_flag_state(*flag_name*, *flag_state*)
Sets a beam flag to a specific state.

Parameters

- **flag_name** (*str*) – The one character flag name listed in the settings (i.e. '1' or '2')
- **flag_state** (*str*) – The one character state string listed in the settings (i.e. 'r' or 'g')

update_all_states()
Queries all flag states and updates this Instruments internal flag states. Returns if any internal state has changed.

Returns true if any state has changed

Return type bool

Testing

In order to help and guide the developer we have created some tests for the specific functionality of each level in the code. The idea is that every time some code is developed, a test is also written so it can be used to ensure the correct behaivour of the code.

We have some basic unit tests in the folder unit_tests. So far we have the following tests

Unit tests

Test LCC25 controller

This class aims to unit_test the correct behaviour of the controller class: LCC25.py

If you have changed something in the controller layer, you should check that the functionalities of if are still running properly by running this class and adding a method to unit_test the new methods in the controller, if any.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.unit_test.lcc_controller.**UTestLcc**(*settings*)
Class to unit_test the LCC25 controller.

finalize()
closes connection

test_freq()
Test the freq command

test_mode()
Test the mode methods

test_output()
Test the output state

test_voltage()
unit_test setter and getter for the voltage1

Test Agilent33522A controller

This class aims to unit_test the correct behaviour of the controller class: agilent33522A.py

If you have changed something in the controller layer, you should check that the functionalities of if are still running properly by running this class and adding a method to unit_test the new methods in the controller, if any.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.unit_test.agilent33522A_controller.**UTestAgilent33522A**(*settings*)
Class to unit_test the LCC25 controller.

finalize()
closes connection

test_all_amplitudes()
To test all the amplitude related setters and getters

test_amplitude()
unit_test setter and getter for the amplitude (Vpp)

test_enable_output()
Test the enable output getter and setter

test_freq()
Test the getter and setter for the frequency command

test_mode()
Test the mode methods

Test variable waveplate instrument

This class aims to unit_test the correct behaviour of the instrument class: variable_waveplate

If you have changed something in the controller and or instrument layer, you should check that the functionalities of if are still running properly by running this class and adding a method to unit_test the new methods in the instrument, if any.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.unit_test.variable_waveplate_instrument.**UTestVariableWaveplate**(*settings*)
Class to unit_test the LCC25 controller.

finalize()
closes connection

test_freq()
Test the freq command

test_mode()
Test the mode methods

test_output()
Test the output state

test_voltage()
unit_test setter and getter for the voltage1

Test FunGen instrument

This class aims to unit_test the correct behaviour of the instrument class: fun_gen

If you have changed something in the controller and or instrument layer, you should check that the functionalities of if are still running properly by running this class and adding a method to unit_test the new methods in the instrument, if any.

NOTE: This is still not implemented as a class, it just runs commands using the instrument. The assertion part has to be done, still.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

class hyperion.unit_test.fun_gen_instrument.**UTestFunGen**(*settings*)
Class to unit_test the FunGen instrument.

finalize()
closes connection

Tools

We also have some useful tools that are used around the whole project. We list them here.

Array Tools

Here we group some array tools that we use in the project.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

hyperion.tools.array_tools.**array_from_pint_quantities**(*start*, *stop*, *step=None*, *num=None*)
Generates an array from pint values. Use either step or num to divide the range up in steps. (If both are specified, step is used) Using num works similar to numpy.linspace(start, stop, num). Using step works somewhat similar numpy.arange(start, stop, step). Modifications are that the sign of step will be interpreted automatically. And the issue of a missing endpoint due to tiny floating point errors is mitigated. It returns a numpy array and the pint unit.

Parameters

- **start** (*pint.quantity*) – The start value of the array
- **stop** (*pint.quantity*) – The end value for determining the array
- **step** (*pint.quantity*) – The stepsize between points
- **num** (*int*) – Number of points generate

Returns a, b

Return type (numpy.array, pint.unit)

hyperion.tools.array_tools.**array_from_settings_dict**(*sweep_dict*)

Wrapper around array_from_string_quantities(). sweep_dict should contain 'start' and 'stop' key. And either 'step' or 'num' key. The values may have units (that can be interpreted by pint). See array_from_string_quantities() and array_from_pint_quantities() for further details.

Parameters **sweep_dict** – Dictionary containing start, stop and step or num keys

Returns (numpy.array, pint.unit)

hyperion.tools.array_tools.**array_from_string_quantities**(*start*, *stop*, *step=None*, *num=None*)

Wrapper around array_from_pint_quantities() that converts string arguments to pint quantities. Arguments start, stop and step should be strings, num could be integer (or string of integer). See array_from_pint_quantities() for further details.

Saving tools

This is a collection of useful methods related to saving data and metadata used along hyperion.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

hyperion.tools.saving_tools.**create_filename**(*file_path*)

Creates the filename, so all the methods point to the same folder and save with the same name. The output does not include the extension but the input does.

Parameters **filename** (*string (path)*) – config filename complete path (INCLUDING the extension)

Returns filename with a number appended so it would not be overwritten.

Return type string

hyperion.tools.saving_tools.**name_incrementer**(*basename*, *list_of_existing*, *separator='_'*, *fill_zeros=0*, *use_number_for_first=None*, *only_larger_number=True*)

This function is meant to avoid rewriting existing files.

Parameters

- **basename** – The basename. May include extension.
- **list_of_existing** – List of names to avoid
- **separator** – Optional separator between basename and number, DEFAULT is '_'
- **fill_zeros** – If you set this to 4, numbers will look like 0012, 0013. Special case 0 (DEFAULT) will match the longest one in list_of_existing

- **use_number_for_first** – 0, 1, None. If the name does not occur yet it will get 0, 1 or no sufix, DEFAULT is None
- **only_larger_number** – If name_1 and name_3 exist and only_larger_number is True DEFAULT, name_4 will be suggested, otherwise name_2

Returns The suggested name.

hyperion.tools.saving_tools.**read_netcdf4_and_plot_all**(*filename*)
Reads the file in filename and plots all the detectors

hyperion.tools.saving_tools.**save_metadata**(*file_path*, *properties*)
Saves the the properties in a yml file at the file file_path

type file_path str

param properties dictionary containing the me :param file_path: complete filepath to the location to be saved.

tadata

type properties dict

hyperion.tools.saving_tools.**save_netCDF4**(*filename*, *detectors*, *data*, *axes*, *axes_name*, *errors=None*, *extra_dims=None*, *description=None*)
This function saves the data in a netCDF4 format, including units and the axes corresponding to the data. For more info about the package used, please see http://unidata.github.io/netcdf4-python/netCDF4/index.html. For info on the format itself, refer to: https://www.unidata.ucar.edu/software/netcdf/docs/index.html

Parameters

- **detectors** (*list of strings*) – list of the detectors used
- **data** (*list of numpy ndarrays of pint quantities*) – list with the actual data corresponding to each detector
- **axes** – corresponding coordinates for the data. the dimension of the i element

in the list has to match the dimension of the i-th dimension of data. :type axes: list of vectors of pint quantity :param axes_name: the name of each of the axes :type axes_name: list of strings :param errors: measured errors for each of the detectors. It has to have the same dimensions as data. :type errors: list of numpy ndarrays of pint quantities :param extra_dims: A dict containing extra values fixed and all the same for the set. :type extra_dims: dict :param description: an extra descriptive message can be put here. :type description: string

hyperion.tools.saving_tools.**yaml_dump_builtin_types_only**(*object*, *stream=None*, *mode='remove'*, *replace_with='_invalid_'*, *dump=True*)
A replacement for yaml.dump that skips object that can't be dumped safely. Dictionaries and lists are searched recursively. It has 3 modes how to handle invalid entries: remove: removes the entry repr: replaces the object with object.__repr__() replace: replaces with the value in replace_with To return the modified object instead of dumping it, set dump to False. (Note, in case of DefaultDict or ActionDict it only stores the main dict).

Parameters

- **object** – The object to dump
- **stream** – The stream to save the yaml dump (default: None)
- **(str)** (*mode*) – 'remove' (default), 'repr', 'replace'
- **replace_with** – What to replace the invalid entry with (default '_invalid_')

- **(bool)** (*dump*) – Dumps to stream when True, returns modified object when False. (default: True)

Returns

UI tools

We group many functions useful for when building GUIS.

copyright by Hyperion Authors, see AUTHORS for more details.

license BSD, see LICENSE for more details.

hyperion.tools.ui_tools.**add_pint_to_combo**(*comboBox_units*, *manual_list=None*)

When GUI has a QComboBox with units, this function can convert the display texts of the units to pint units and stores them inside the combobox object. (Run this function once). It is possible to manually specify the list to store, but it's safer to try automatic conversion.

Parameters

- **comboBox_units** (*QComboBox*) –
- **manual_list** (*list of pint units*) – OPTIONAL list of pint units corresponding to the display units

hyperion.tools.ui_tools.**pint_to_spin_combo**(*pint_quantity*, *doubleSpinBox*, *comboBox_units*)

When a GUI has a combination of a Q(Double)SpinBox and a QComboBox that hold the numeric value and the unit respectively, this function can be used to easily put a pint quantity into that combination. It is strongly recommended to use add_pint_to_combo(comboBox_units) before using this function! Complementary function is spin_combo_to_pint_apply_limits()

Parameters

- **pint_quantity** (*pint quantity*) – the pint quantity to write into the gui objects
- **doubleSpinBox** (*QDoubleSpinBox (Maybe QSpinBox also works. Not tested)*) – the QDoubleSpinBox (or QSpinBox?) that holds the numeric value
- **comboBox_units** (*QComboBox*) – the QComboBox that holds the units

hyperion.tools.ui_tools.**spin_combo_to_pint_apply_limits**(*doubleSpinBox*, *comboBox_units*, *pint_lower_limit=None*, *pint_upper_limit=None*)

When a GUI has a combination of a Q(Double)SpinBox and a QComboBox that hold the numeric value and the unit respectively, this function can be used to convert the combined values to a pint quantity. In addition it applies limits if they are specified. Typically you'll make one function limit_and_apply_X in your gui code, and connect both

```
>>> doubleSpinBox.valueChanged.connect(limit_and_apply_X)
>>> comboBox_units.currentIndexChanged.connect(limit_and_apply_X)
```

to this function. Inside limit_and_apply_X() you would use this function spin_combo_to_pint_apply_limits() to apply limits and convert it to a pint quantity

It is strongly recommended to use add_pint_to_combo(comboBox_units) before using this function! Complementary function is pint_to_spin_combo()

Parameters

- **doubleSpinBox** – Q(Double)SpinBox that holds the numeric value

- **comboBox_units** – QComboBox that holds the units
- **pint_lower_limit** – OPTIONAL pint quantity for the lower limit to apply
- **pint_upper_limit** – OPTIONAL pint quantity for the upper limit to apply

Returns pint quantity

View and GUI

Here we group the information about the higher layer of complexity in our code: the GUI. Building a GUI requires time and in many cases is not needed.

Hyperion Core Utilities

Explanation of how to use logging

Short version:

There is a single logging manager that you import in your files. This logging manager can create a logger objects (which you use in your files and classes to do log-prints like logger.info('bla'). When creating the logger object, a stream handler (writes to screen) and a file handler are passed with it. These take care of the layout, the level (e.g. whether lower levels like debug or info are printed), and which file to write to. You can choose to disable the stream or file handler, modify their layout and levels individually. The stream handler colors can be turned on or off and the color format can be modified.

Full explanation:

In hyperion.core the class LoggingManager is defined. It is a "Singleton" class meaning that all instantiations of this class are actually one and the same object.|br| In hyperion.core one object is already instantiated by the name of logman.|br| In hyperion __init__.py this is imported under the alias name logging.|br| Because it's a singleton class the following approaches of importing the logging manager object are all equivalent:

- from hyperion import logging
- from hyperion.core import logman as logging
- from hyperion.core import logman
- from hyperion.core import LoggingManager import hyperion log = LoggingManager(hyperion.log_path, 'my_log_name.log')
- import hyperion.core log = hyperion.core.LoggingManager()

In all cases above logging, logman and log refer to the same single object of class LoggingManager. For the rest of the following explanation logman is used, but if you're modifying an existing file that used logging, you could remove the 'import logging' and replace it with 'from hyperion import logging'. Note, the optional arguments are default_path and default_name for logging file. LoggingManager has two optional arguments default_path and default_name, which are used as default values for creating the log file. By default, the path is hyperion.log_path and the name is 'hyperion.log' In your own project you could change those if you like.

Now, to use logging in your module (file) or in your class, you have to create a logger: logger = logman.getLogger(__name__) Note that instead of the usual module name __name__ you could hypothetically put in any string. Note that as a shorthand you can also do directly: logger=logman(__name__) When the logger is created, by default, both the stream handler and the file handler are passed with it. These take care of printing to the screen

and to a file. If you don't want to add both, you could omit adding them by setting the optional keyword arguments add_stream or add_file to False. e.g.; logger = logman.getLogger(__name__, add_file=False)

Before creating a logger object, you can:

- Change the default path or name of the logfile in the stream_handler: logman.default_path = 'd:' logman.default_name = 'my_project.log'
- Set the level of the default stream or file handler in the logging manager (defaults are DEBUG) logman.stream_level = logman.WARNING # note: you could also pass in the string 'WARNING' logman.file_level = 'INFO' # note: you could also pass in logman.INFO (Note that changing the level in the manager after a logger object is created is likely to also affect that logger. This depends if the handler was modified in the meantime. If it wasn't, then it's the same object.)
- Change the default stream or file handler in the logging manager Change layout, file name, colors. See below.
- Enable/disable whether the stream of the file handler is passed when a logger object is created (default is True). logman.enable_stream = False logman.enable_file = True

After creating a logger object you can still:

- Add handlers logman.remove_stream_handler(my_handler) logman.remove_file_handler(my_handler)
- Or remove handlers logman.add_stream_handler(my_handler) logman.add_file_handler(my_handler)
- And you can modify the logging level of its handers. Just be aware that if the handlers in the manager weren't modified in the meantime, changing the level of one logger may change the level of others as well. logman.set_logger_stream_level(my_handler, 'WARNING') logman.set_logger_file_level(my_handler, logman.INFO)

Finally, explanation of setting up the handlers:

To create (or replace) the handlers use: logman.set_stream(optional arguments) logman.set_file(optional arguments) All arguments are optional (they have a default value if omitted) The arguements in both set_stream and set_file are:

- **level If omitted, the default value is used** (default value can be set by logman.stream_level / logman.file_level)
- **reduce_duplicates Enables a Filer that detect repeated log comments (that were in a loop) and reduces the number** of lines printed. Defaults to True
- **compact A float between 0 and 1. A measure of how long or compact a line will be. 0 Means full length.** As the value of compact is increased the lines will become shorter. At 1 the line will be most compact and it will be cut off at length maxwidth. Default value for set_stream is 0.5, default value for set_file is 0
- maxwidth See description of compact. Default value is 119

Arguments only in set_stream:

- color A bool that determines whether to use colors when printing levels names. Defaults to True
- **color_scheme A string indicating color_scheme. Possible values are: bright, dim, mixed, bg, universal** If the logger is used in Spyder it diverts to a modified scheme. To manually select different schemes for both non-Spyder and Spyder do something like color_scheme=('mixed', 'spy_bright') Scheme 'universal' will be readable and most similar on Spyder, Pycharm, PyCharm-darcula, regular black terminal and blue PowerShell terminal. But it's not ver esthetically pleasing, so the default is ('bright', 'spy_bright')
- All other keyword arguments are passed into logging.StreamHandler().

Arguments only in set_file:

- **pathname The path or only filename or full file-path. If path is not included it will use the default** path. If name is not included it will use the default name)
- All other keyword arguments are passed into logging.handlers.RotatingFileHandler(). maxBytes will default to 5MB and backupCount will default to 9

example code 1:

```
from hyperion import logging

class A:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.logger.info('object of class A created')

if __name__=='__main__':
    logging.enable_file = False
    logging.stream_level = 'INFO'
    logging.set_stream(compact=0.4, color=False)
    a = A()
```

example code 2:

```
from hyperion.core import logman

class A:
    def __init__(self):
        logger = logman(__name__, add_stream=False)      # no stream logging at all␣
↪for this class
        logger.info('object of class A created')

class B:
    def __init__(self):
        self.logger = logman(__name__)
        self.logger.info('object B: 1 - info')

        # Temporarily change logging level (note that this may affect other loggers␣
↪as well)
        lvl = logman.get_logger_stream_level(self.logger)
        logman.set_logger_stream_level(self.logger, 'WARNING')
        self.logger.info('object B: 2 - info')
        self.logger.warning('object B: 2 - warning')
        logman.set_logger_stream_level(self.logger, lvl)

        self.logger.info('object B: 3 - info')

class C:
    def __init__(self):
        self.logger = logman(__name__)
        self.logger.info('object C: 1 - info')

        # Temporarily remove handler from logger
        h = logman.remove_stream_handler(self.logger)       # storing it in h is␣
↪optional
```

(continues on next page)

(continued from previous page)

```
        self.logger.info('object C: 2 - info')
        self.logger.warning('object C: 2 - warning')
        # To restore the exact handler:
        self.logger.addHandler(h)
        # It's also possible to add the handler from the manager:
        # logman.add_file_handler(self.logger)

        self.logger.info('object C: 3 - info')


if __name__=='__main__':
    logging.default_name = 'my_project.log'
    logging.set_stream(compact=0.4, color=False)
    a = A()
    b = B()
    c = C()
```

class hyperion.core.**ANSIcolorFormat**(*enable=True*)

Class used as a wrapper function to apply colors to console prints. Arguments are the string followed by one or more color 'decorators': First letter of basic colors: r, g, y, b, m, c, w for white, k for black. Letter Preceded by an l means a lighter/brighter color (if supported). Letter(s) preceded by an underscore '_' means background color. Notes on PyCharm: 'emph' creates bold text. Darcula mode changes colors into bland "pastel-like" colors and inverts pure white and black. Notes on general command/prompt window: 'emph' turns dark text to bright. Notes on Spyder: Does not support "light" colors with an "l", 'emph'/'norm' turns both text and background simultaneously to bright/dim.

The 'decorators' can be passed as multiple arguments or a tuple or a list. A single string of ansi color code is also accepted (e.g. '1;46;31') If no additional arguments or the argument None is passed it returns the message without adding ansi color codes. The enabled property allows for toggling all color action of this object at once. The class method disable_all() allows to disable all color printing of all objects.

Note: It's also possible to do from hyperion.core import ansicol Then you're using the same object (i.e. toggling that object will influence all)

Example usage: from hyperion.core import ANSIcolorFormat ansicol = ANSIcolorFormat() print(ansicol('Hello World','emph','r','_y')) print(ansicol('Hello World')) ansicol.enabled = False print(ansicol('Hello World','emph','r','_y'))

static disable_all(*boolean*)

Class Method (with boolean input) to disable all color printing of all ANSIcolorFormat objects at once. Useful if your system can't deal with ANSI color codes. Note that this will overrule the enabled state of individual objects. Being a class method this can be run both on an object and directly on the class, like ANSIcolorFormat.disable_all(False)

enabled

Boolean property to get or set whether this ANSIcolorFormat object is enabled. If enabled is false it will not print colors. Note that the class disabled_all state overrules this object state. To enable you may also have to do .disable_all(False) on the object or on the class.

class hyperion.core.**CustomFormatter**(*compact=0.0*, *maxwidth=None*, *color=False*, *color_scheme=None*)

Custom format for log-prints. Adds a lot of information (time, module, line number, method, LEVEL, message) Setting the compact parameter to a value larger than 0 shrinks the date, module and method. Setting it to 1 (maximum) assures the length is maxwidth characters or less. The possible regular color schemes are: bright, dim, mixed, bg, universal. The optional additional secondary color schemes for Spyder are: spy_bright, spy_dim, spy_mixed, spy_bg, spy_universal. Specifying no color_scheme defaults to bright (and spy_bright if Spyder is detected).

Parameters

- **compact** – float from 0 for full length, to 1 for very compact (defaults to 0)
- **maxwidth** – integer indicating max line width when compact is 1. None (default) uses 119.
- **color** – boolean indicating if ANSI colors should be used (defaults to False)
- **color_scheme** – string or tuple/list of two strings (defaults to bright / spy_bri)

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class hyperion.core.**DuplicateFilter**(*name=''*)
Adding this filter to a logging handler will reduce repeated log-prints

filter(*record*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class hyperion.core.**LoggingManager**(*default_path='/home/docs/checkouts/readthedocs.org/user_builds/nanooptics-hyperion/envs/latest/lib/python3.7/site-packages/logs'*, *default_name='hyperion.log'*)
LogginManager class. This is a "Singleton" class which means that all of its instantiatons refer to one and the same object. If one tries to create a second object, the original object is returned. For a more elaborate explanation on how to use, see beginning of page.

Variables

- **enable_stream** – (bool) If true, the stream handler is passed to newly created logger objects (default is True)
- **enable_file** – (bool) If true, the file handler is passed to newly created logger objects (default is True)
- **default_path** – (str) Default path for logfile when file handler is created without specifying path (default hyperion.log_path)
- **default_name** – (str) Default filename for logfile when file handler is created without specifying filename (default 'hyperion.log;)

add_file_handler(*logger*)
Add file_handler to an existing logger object.

Parameters **logger** – a logger object

add_stream_handler(*logger*)
Add stream_handler to an existing logger object.

Parameters **logger** – a logger object

file_level
Property to read and set the level of the current file handler. When setting it also updates the default level.

getLogger(*name*, *add_stream=None*, *add_file=None*)
Returns logger object.

Parameters

- **name** – (str) Name, usually the module name. i.e. __name__
- **add_stream** – (bool or None) add the streamhandler. None (default) uses object.enalbe_stream
- **add_file** – (bool or None) add the streamhandler. None (default) uses object.enalbe_stream

Returns logger object

get_logger_file_level (*logger*)
Get the level of the (first) file handler of an existing logger object.

Parameters **logger** – existing logger object

get_logger_stream_level (*logger*)
Get the level of the (first) stream handler of an existing logger object.

Parameters **logger** – existing logger object

remove_file_handlers (*logger*)
Remove stream_handlers from an existing logger object.

Parameters **logger** – a logger object

Returns the removed file handler object (or None)

remove_stream_handler (*logger*)
Remove stream_handlers from an existing logger object.

Parameters **logger** – a logger object

Returns the removed stream handler object (or None)

set_file (*pathname=None*, *level=None*, *compact=0*, *reduce_duplicates=True*, *maxwidth=None*, *maxBytes=5242880*, *backupCount=9*, ***kwargs*)
Sets (replaces) the file handler in the logging manager object.

Parameters

- **pathname** – path or filename or full file-path (if only name or path is given, the other will use the default value)
- **level** – logging level. If omitted, default is used. To change default see file_level
- **compact** – see CustomFormatter (defaults to 0)
- **maxwidth** – see CustomFormatter
- **reduce_duplicates** – (bool) (defaults to True)
- **maxBytes** – see logging.handlers.RotatingFileHandler() (defaults to 5 * 1024 * 1024)
- **backupCount** – see logging.handlers.RotatingFileHandler() (defaults to 9)
- ****kwargs** – additional keyword arguments are passed into logging.handlers.RotatingFileHandler()

set_logger_file_level (*logger*, *level*)
Change level of the file handler of an existing logger object. Note that this may affect other logger object, because they may share the same file handler object.

Parameters

- **logger** – existing logger object

- **level** – string like 'WARNING' or level like logman.WARNING

set_logger_stream_level(*logger*, *level*)
Change level of the stream handler of an existing logger object. Note that this may affect other logger object, because they may share the same stream handler object.

Parameters

- **logger** – existing logger object
- **level** – string like 'WARNING' or level like logman.WARNING

set_stream(*color=True*, *level=None*, *compact=0.5*, *reduce_duplicates=True*, *maxwidth=None*, *color_scheme=None*, ***kwargs*)
Sets (replaces) the stream handler in the logging manager object.

Parameters

- **color** – (bool) whether to print levelnames with color
- **level** – logging level. If omitted, default is used. To change default see file_level
- **compact** – see CustomFormatter (defaults to 0)
- **maxwidth** – see CustomFormatter
- **reduce_duplicates** – (bool) (defaults to True)
- **color_scheme** – (str or (str,str)) color scheme to use, see above for possible values
- ****kwargs** – additional keyword arguments are passed into logging.StreamHandler()

Returns

stream_level
Property to read and set the level of the current stream handler. When setting it also updates the default level.

class hyperion.core.**Singleton**
Metaclass to use for classes of which you only want one instance to exist. use like: class MyClass(ParentClass, metaclass=Singleton): If one tries to create a second object, the original object is returned.

1.5 How to install

For installation instructions, please refer to the readme.md in the root folder of the project.

1.6 Authors

Naturally, this list is in continuous change since we have many people contributing to this project. We try to keep it updated so new developers can contact others working in the project to benefit from everyone strengths.

This version for the software is strongly based on the ideas presented in the book "Python For The Lab".

So far, we have contributed to this project:

- Martin Caldarola (m.caldarola@tudelft.nl)
- Aron Opheij (a.opheij@tudelft.nl)
- Thomas Bauer
- Irina Komen

- Marc Noordam
- Thijs Van Gogh
- Ariel Komen

Python Module Index

h

Index

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z