PyExpLabSys Documentation

Release 1.5

CINF

Dec 06, 2018

Contents

This page serves as documentation for the software activities at CINF. Of interest to the public are hardware drivers for experimental equipment and data logging clients but the page also contain documentation for setup specific code.

Contents:

CHAPTER 1

Overview

This page contains different overviews of the PyExpLabSys archive and can work as an entry point for a new user.

Section *Project Overview* contains a short explanation of the different components that PyExpLabSys consist of and the source code and documentation entries that are relevant for that part.

The table in section *Module Overview* contains an overview of all the modules in PyExpLabSys. The overview consist of a short description (the first line of the module docstring) and its Python 2/3 support status.

PyExpLabSys strive to support Python version 2.7 and >=3.3. See the section *Python 3 support* about Python 3 support.

1.1 Project Overview

This section will explain how the different components of PyExpLabSys (and its sister project cinfdata) fits together.

The overall structure is illustrated in the *overview figure*. On the aquisition level, there is a number of machines which could be e.g. Raspberry Pi's or laboratory PC's, which will use different parts of PyExpLabSys depending on its purpose.

Machine 1 e.g. is used to parse already aquired data from local data files (*File parsers*) and send that data to the database (*Database Savers*). Machine 2 aquires data via an equipment drivers (*Drivers*) and makes it available to other machines on the network (*Sockets*) and uploads it to the database (*Database Savers*). Like machine 2, machine 3 also aquires data directly (*Drivers*), but it also makes use of the data that machine 2 makes available (*Sockets*).

On the server level there is a MySQL servers to which all data is saved and a webserver. The webserver runs a webpage called "Cinfdata" (source code), which is used for data presentation and light data treatment.

On the user level, users will commonly access the data via the Cinfdata website in a browser on the desktop or on a smart phone. To get the data locally to do more extensive data treament, it is possible to fetch the data directly from MySQL server or to use an export function on the website.

In the following, each of these software components will be described in a little more detail.

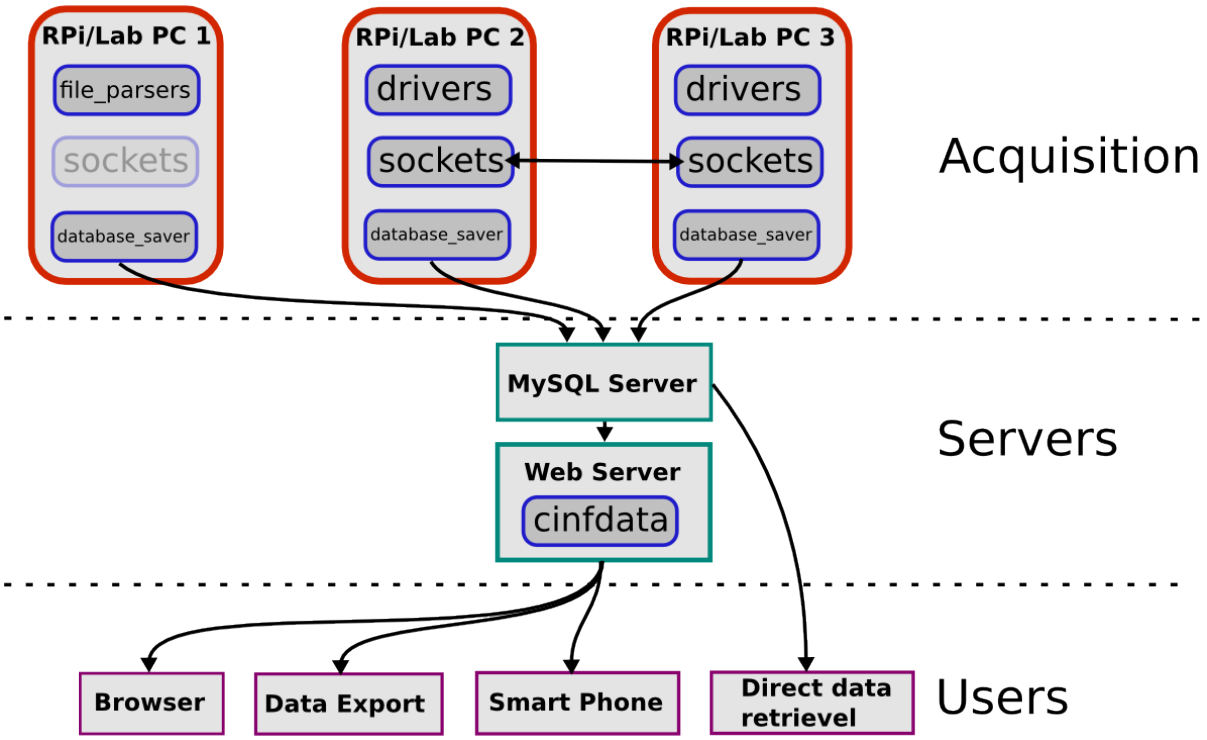


Fig. 1: Figure illustrating how the different software components (boxes with blue edge) of PyExpLabSys (and cinfdata) fits together.

1.1.1 Drivers

At this point PyExpLabSys contains a reasonable amount of drivers (46 files, 81 classes May-16) for general purpose equipment (data cards, temperature readout etc.) and for equipment related specifically to the vacuum lab we work in (pressure gauge controllers, mass spectrometers etc).

The source code for the drivers are in the drivers folder, in which the file names are either manufacturer and model or just the manufacturer.

The documentation for the drivers are divided into two sections *Hardware Drivers* and *Hardware Drivers Autogenerated Docs Only*. The latter is the group for which there is only API documentation auto generated from the source code and the former are the drivers that has more specific documentation with example usage etc.

1.1.2 File parsers

PyExpLabSys also contains a small number of parsers for custom file formats. The source code for these are in the file_parsers folder and the documentation is in the *File parsers* section.

1.1.3 Database Savers

The database savers are some of the more frequently used classes in PyExpLabSys. Quite simply, they abstract away; the database layout, the SQL and the queuing of data ofloading (to prevent loosing data in the event of data loss). The source code for the database savers are in database_saver module in the common sub-package. The documentation is located at *The database_saver module*.

1.1.4 Sockets

The sockets are another set of very important and highly used classes in PyExpLabSys. Most of the sockets are socket servers, which mean that they accept UDP requests and serves (or accepts) data. These are essentially used as network variables, by either exposing a measurement on the network or accepting input. A final type of socket is the LiveSocket which is used to live stream data to a live streaming proxy server. Furthermore, all sockets also expose system (health) information to the network. The code for the sockets are found in the sockets module in the common sub-package. The documentation is located at *The sockets module*.

1.1.5 Apps

The apps section contains stand alone programs which can be used directly and not only as service functions of larger programs. Typically the apps sports text based guis and several apps also have ambitions for graphical guis. Notable examples of apps are: socket supervisor This very simple app continously probes a port (typically a socket) and reports back if the socket is running correctly.

Mass Spectrometer Controls a number of Pfeiffer mass spectrometers. The app includes support for an infinite amount of meta-channels wich is pulled in via sockets.

Picture Logbook A graphical logbook system based on a barcode reader and a camera. The user of the equipment logs in before use and out after use. A picture of the equipment is acquired at login and logout. Also contains support for an external screen wich shows the currently logged in user.

Turbo pump controler Logs and controls Pfeiffer turbo pumps. A number of important parameters are shown in a terminal and the values are also availble as live sockets.

1.1.6 Misc.

Besides from the items listed above PyExpLabSys contains a number of little helpers. The `PyExpLabSys.common.utilities` (code, *doc*) module contains a convenience function to get a logger, that is configured that way that we prefer, including email notification of anything warning level or above.

1.2 Python 3 support

We love Python 3. Unfortunately we are not hired to make software, but to keep a lab running. This means that modules are only ported to Python 3, when it is either convinient or we are touching the code anyway.

1.3 Module Overview

PyExpLabSys.combos (Python 2 only[1])

- This module contains socket, database saver and logger heuristic combinations

---

PyExpLabSys.settings (Python 2 only[1])

- This module contains the modules used for settings for PyExpLabSys

---

PyExpLabSys.common.plotters_backend_qwt (Python 2 only[1])

[1] For these modules the Python 2/3 status is not indicated directly in the source code file and so the status is inferred.

- This module contains plotting backends that use the PyQwt library

---

PyExpLabSys.common.system_status (Python 2 only[1])

- This module contains the SystemStatus class

PyExpLabSys.common.utilities (Python 2 only[1])

- This module contains a convenience function for easily setting up a logger with the `logging` module.

PyExpLabSys.common.value_logger (Python 2 and 3)

- Read a continuously updated values and decides whether it is time to log a new point

---

PyExpLabSys.common.valve_control (Python 2 and 3)

- This module implements the necessary interface to control a valve box using standard gpio commands

---

PyExpLabSys.common.socket_clients (Python 2 only[1])

- This file implements Python clients for the DateDataPullSocket

---

PyExpLabSys.common.microreactor_temperature_control (Python 2 and 3)

- Common code for microreactor heaters

PyExpLabSys.common.supported_versions (Python 2 and 3)

- Functions used to indicate and check for supported Python versions

---

PyExpLabSys.common.analog_flow_control (Python 2 only[1])

- Control app for analog pressure controller on sniffer setup

---

PyExpLabSys.common.loggers (Python 2 only[1])

- This module contains convinience classes for database logging. The classes implement queues to contain the data before of loading to the database to ensure against network or server problems.

PyExpLabSys.common.flow_control_bronkhorst (Python 2 and 3)

- Common code for Bronkhorst boxes

PyExpLabSys.common.plotters (Python 2 only[1])

- This module contains plotters for experimental data gathering applications. It contains a plotter for data sets.

PyExpLabSys.common.decorators (Python 2 only[1])

- This module contains general purpose decorators

PyExpLabSys.common.text_plot (Python 2 only[1])

- GNU plot based plots direcly into a curses window

PyExpLabSys.common.chiller_reader (Python 2 and 3)

- Module for monitoring a polyscience chiller

PyExpLabSys.common.sockets (Python 2 and 3)

- The sockets module contains various implementations of UDP socket servers (at present 4 in total) for transmission of data over the network. The different implementations are tailored for a specific purposes, as described below.

PyExpLabSys.common.database_saver (Python 2 and 3)

- Classes for saving coninuous data and data sets to a database

PyExpLabSys.common.massspec.channel (Python 2 only[1])

- This module contains the implementation of a general purpose mass spectrometer channel

PyExpLabSys.common.massspec.test (Python 2 only[1])

- Module to test the mass spec common components

PyExpLabSys.common.massspec.qt (Python 2 only[1])

- Module that contains QT widgets for mass spectrometer channels and channel lists

PyExpLabSys.drivers.pfeiffer_turbo_pump (Python 2 and 3)

- Self contained module to run a Pfeiffer turbo pump including fall-back text gui and data logging.

PyExpLabSys.drivers.brooks_s_protocol (Python 2 and 3)

- Driver for Brooks s-protocol

PyExpLabSys.drivers.honeywell_6000 (Python 2 and 3)

- Driver for HIH6000 class temperature and humidity sensors

PyExpLabSys.drivers.stmicroelectronics_ais328dq (Python 2 and 3)

- Driver for STMicroelectronics AIS328DQTR 3 axis accelerometer

---

PyExpLabSys.drivers.four_d_systems (Python 2 and 3)

- Drivers for the 4d systems displays

PyExpLabSys.drivers.specs_XRC1000 (Python 2 only[1])

- Self contained module to run a SPECS sputter gun including fall-back text gui

---

PyExpLabSys.drivers.omega_D6400 (Python 2 and 3)

- Driver for Omega D6400 daq card

PyExpLabSys.drivers.mks_937b (Python 2 and 3)

- Driver for MKS 937b gauge controller

PyExpLabSys.drivers.freescale_mma7660fc (Python 2 and 3)

- Driver for AIS328DQTR 3 axis accelerometer

PyExpLabSys.drivers.mks_925_pirani (Python 2 and 3)

- Driver for MKS 925 micro pirani

PyExpLabSys.drivers.omega_cni (Python 2 and 3)

- This module contains drivers for equipment from Omega. Specifically it contains a driver for the ??? thermo couple read out unit.

PyExpLabSys.drivers.rosemount_nga2000 (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.drivers.innova (Python 2 only[1])

- Driver for the Innova RT 6K UPS

PyExpLabSys.drivers.agilent_34972A (Python 2 and 3)

- Driver class for Agilent 34972A multiplexer

PyExpLabSys.drivers.edwards_nxds (Python 2 and 3)

- Driver for Edwards, nXDS pumps

PyExpLabSys.drivers.omegabus (Python 2 and 3)

- Driver for OmegaBus devices

PyExpLabSys.drivers.omega_cn7800 (Python 2 only[1])

- Omega CN7800 Modbus driver. Might also work with other CN units

PyExpLabSys.drivers.vogtlin (Python 2 and 3)

- Minimal MODBUS driver for the red-y smart - meter GSM, - controller GSC, - pressure controller GSP and - back pressure controller GSB.

PyExpLabSys.drivers.xgs600 (Python 2 and 3)

- Driver class for XGS600 gauge controll

PyExpLabSys.drivers.pfeiffer (Python 2 only[1])

- This module contains drivers for the following equipment from Pfeiffer Vacuum:

PyExpLabSys.drivers.dataq_comm (Python 2 and 3)

- Driver for DATAQ dac units

PyExpLabSys.drivers.polyscience_4100 (Python 2 and 3)

- Driver and test case for Polyscience 4100

PyExpLabSys.drivers.keithley_2700 (Python 2 and 3)

- Simple driver for Keithley Model 2700

PyExpLabSys.drivers.scpi (Python 2 and 3)

- Implementation of SCPI standard

PyExpLabSys.drivers.NGC2D (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.drivers.stahl_hv_400 (Python 2 and 3)

- Driver for Stahl HV 400 Ion Optics Supply

PyExpLabSys.drivers.omron_d6fph (Python 2 and 3)

- Hint for implementation found at http://forum.arduino.cc/index.php?topic=285116.0

PyExpLabSys.drivers.keithley_smu (Python 2 and 3)

- Simple driver for Keithley SMU

PyExpLabSys.drivers.mks_g_series (Python 2 and 3)

- Driver for MKS g-series flow controller

PyExpLabSys.drivers.vivo_technologies (Python 2 only[1])

- Driver for a Vivo Technologies LS-689A barcode scanner

PyExpLabSys.drivers.deltaco_TB_298 (Python 3 only)

- Driver with line reader for the Deltaco TB-298 Keypad

PyExpLabSys.drivers.mks_pi_pc (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.drivers.srs_sr630 (Python 2 and 3)

- Driver for Standford Research Systems, Model SR630

PyExpLabSys.drivers.dataq_binary (Python 2 and 3)

- DataQ Binary protocol driver

PyExpLabSys.drivers.bio_logic (Python 2 only[1])

- This module is a Python implementation of a driver around the EC-lib DLL. It can be used to control at least the SP-150 potentiostat from Bio-Logic under 32 bit Windows.

PyExpLabSys.drivers.fug (Python 2 only)

- Driver for “fug NTN 140 - 6,5 17965-01-01” power supply Communication via the Probus V serial interface.

PyExpLabSys.drivers.cpx400dp (Python 2 and 3)

- Driver for CPX400DP power supply

PyExpLabSys.drivers.epimax (Python 2 and 3)

- Driver for the Epimax PVCi process vacuum controller

PyExpLabSys.drivers.bronkhorst (Python 2 and 3)

- Driver for Bronkhorst flow controllers, including simple test case

---

PyExpLabSys.drivers.agilent_34410A (Python 2 and 3)

- Driver class for Agilent 34410A DMM

PyExpLabSys.drivers.microchip_tech_mcp3428 (Python 2 and 3)

- Driver for Microchip Technology MCP3428 Analog Input device Calibrated to PR33-13 from ncd.io other products will use different voltage references.

PyExpLabSys.drivers.isotech_ips (Python 2 only[1])

- Driver for ISO-TECH IPS power supply series

PyExpLabSys.drivers.galaxy_3500 (Python 2 and 3)

- Python interface for Galaxy 3500 UPS. The driver uses the telnet interface of the device.

PyExpLabSys.drivers.lascar (Python 2 only[1])

- Driver for the EL-USB-RT temperature and humidity USB device from Lascar

---

PyExpLabSys.drivers.pfeiffer_qmg420 (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.drivers.tenma (Python 2 and 3)

- from __future__ import unicode_literals, print_function

PyExpLabSys.drivers.inficon_sqm160 (Python 2 only[1])

- Driver for Inficon SQM160 QCM controller

PyExpLabSys.drivers.crowcon (Python 2 and 3)

- This module contains a driver for the Vortex gas alarm central

PyExpLabSys.drivers.analogdevices_ad5667 (Python 3 only)

- Driver for the Analog Devices AD5667 2 channel analog output DAC

PyExpLabSys.drivers.kjlc_pressure_gauge (Python 2 and 3)

- Module contains driver for KJLC 3000 pressure gauge

PyExpLabSys.drivers.pfeiffer_qmg422 (Python 2 and 3)

- This module contains the driver code for the QMG422 control box for a pfeiffer mass-spectrometer. The code should in principle work for multiple type of electronics. It has so far been tested with a qme-125 box and a qme-??? box. The module is ment as a driver and has very little function in itself. The module is ment to be used as a sub-module for a large program providing the functionality to actually use the mass-spectrometer.

PyExpLabSys.drivers.stmicroelectronics_l3g4200d (Python 2 and 3)

- Driver for STMicroelectronics L3G4200D 3 axis gyroscope

PyExpLabSys.drivers.specs_iqe11 (Python 2 only[1])

- Self contained module to run a SPECS sputter gun including fall-back text gui

PyExpLabSys.drivers.wpi_al1000 (Python 2 only[1])

- This module implements a driver for the AL1000 syringe pump from World Precision Instruments

PyExpLabSys.drivers.intellemetrics_il800 (Python 2 only[1])

- Driver for IL800 deposition controller

PyExpLabSys.drivers.edwards_agc (Python 2 only[1])

- Driver and simple test case for Edwards Active Gauge Controler

PyExpLabSys.auxiliary.rtd_calculator (Python 2 and 3)

- Calculates temperatures for an RTD

PyExpLabSys.auxiliary.pid (Python 2 and 3)

- PID calculator

PyExpLabSys.auxiliary.tc_calculator (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.thirdparty.cached_property (Python 2 only[1])

- Copy of the cached property project from https://github.com/pydanny/cached-property

---

PyExpLabSys.thirdparty.olefile (Python 2 only[1])

- Launched from the command line, this script parses OLE files and prints info.

PyExpLabSys.apps.mass_finder (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.apps.wind_speed_logger (Python 2 and 3)

- Logs fume hood wind speed

PyExpLabSys.apps.bakeout (Python 2 and 3)

- App to control PW-modulated bakeout boxes

PyExpLabSys.apps.socket_tester (Python 2 only[1])

- A socket tester program for linux

PyExpLabSys.apps.picture_logbook (Python 2 only[1])

- Module to run a graphical logbook of a specific area

PyExpLabSys.apps.edwards_nxds_logger (Python 2 and 3)

- Logger for nXDSni roughing pump

PyExpLabSys.apps.emission_control (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.apps.turbo_logger (Python 2 and 3)

- App to log output from Pfeiffer Turbo Pumps

PyExpLabSys.apps.rampreader (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.apps.socket_supervisor (Python 2 and 3)

- Module to check that local machine is living up to its duties

PyExpLabSys.apps.socket_logger (Python 2 and 3)

- App for logging specific sockets into dateplots

PyExpLabSys.apps.ion_optics_controller (Python 2 and 3)

- Ion Optics Control software

PyExpLabSys.apps.tof.spectrum_plotter (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.apps.tof.fix_mass_axis (Python 2 and 3)

- Program to fix x-axis on TOF-spectra

PyExpLabSys.apps.tof.helper_scripts.lm_test (Python 2 only[1])

- NO DESCRIPTION

PyExpLabSys.apps.bakeoutweb.bakeoutweb (Python 2 only[1])

- Web app for the magnificient bakeout app

PyExpLabSys.apps.qms.qms (Python 2 and 3)

- Mass Spec Main program

PyExpLabSys.apps.qms.mass_spec (Python 2 and 3)

- Mass spec program

PyExpLabSys.apps.qms.qmg_meta_channels (Python 2 and 3)

- Module to perform read-out of meta channels for qms

PyExpLabSys.apps.qms.qmg_status_output (Python 2 and 3)

- Text UI for mass spec program

PyExpLabSys.apps.stepped_program_runner.stepped_program_runner (Python 2 only[1])

- A general stepped program runner

PyExpLabSys.file_parsers.specs (Python 2 and 3)

- This file is used to parse XPS and ISS data from XML files from the SPECS program.

PyExpLabSys.file_parsers.chemstation (Python 2 and 3)

- File parser for Chemstation files

PyExpLabSys.file_parsers.total_chrom (Python 3 only)

- Experimental parser for total_chrom files from Perkin-Elmer GC's

---

PyExpLabSys.file_parsers.omicron (Python 2 only[1])

- File parser for the Omicron "Flattener" format

PyExpLabSys.file_parsers.avantage (Python 2 only[1])

- Test module for Avantage files

PyExpLabSys.file_parsers.avantage_xlsx_export (Python 2 only[1])

- File parser for the Avantage xlsx export format

CHAPTER 2

User Notes

This page has various different notes for using PyExpLabSys.

2.1 Setting up logging of your program

To set up logging of a program, it is possible to simply follow the standard library documentation. However, since many of the programs that uses PyExpLabSys are programs that runs for extended periods without monitoring, a more specific logging setup may be required. E.g. one that makes use of email handlers, so be notified by email in case of errors or warnings.

For that purpose, PyExpLabSys has the *get_logger()* function in the *utilities* module that is a convinience function to set up a logger with one or more of the commonly used log handlers i.e. a terminal handler, a rotating file handler and email handlers. This may be used to things up and running in a hurry.

```
from PyExpLabSys.common import utilities

utilities.MAIL_HOST = 'my.mail.host'
utilities.WARNING_EMAIL = 'email-address-to-use-in-case-of-warnings@log.com'
utilities.ERROR_EMAIL = 'email-address-to-use-in-case-of-error@log.com'

# Returns a logger with terminal and emails handlers per default
LOG = utilities.get_logger('my_program_name')

# A rotating file handler can be added:
LOG = utilities.get_logger('my_program_name', file_log=True)
```

2.2 Activating PyExpLabSys library logging in you program

PyExpLabSys contains quite a few loggers and exposes a few convinience functions in the *utilities* module for listing and activating them. To get a list of loggers that are relevant for the modules that you have im-

ported, you can use either the *get_library_logger_names()* function, which will return you a list or the *print_library_logger_names()* function, which prints them out:

```
from PyExpLabSys.common import sockets
from PyExpLabSys.common.utilities import print_library_logger_names

print_library_logger_names()
```

produces the following output:

```
Current PyExpLabSys loggers
===========================
 * PyExpLabSys.common.sockets.PullUDPHandler
 * PyExpLabSys.common.sockets.DataPushSocket
 * PyExpLabSys.common.sockets.PushUDPHandler
 * PyExpLabSys.settings
 * PyExpLabSys.common.sockets.DateDataPullSocket
 * PyExpLabSys.common.sockets.CallBackThread
 * PyExpLabSys.common
 * PyExpLabSys.common.sockets.CommonDataPullSocket
 * PyExpLabSys.common.sockets
 * PyExpLabSys.common.sockets.DataPullSocket
 * PyExpLabSys
 * PyExpLabSys.common.sockets.LiveSocket
```

To activate a logger use the full path of the logger e.g. `PyExpLabSys.common.sockets.DataPullSocket` and remembers that the loggers are configured as a tree, so activating `PyExpLabSys.common.sockets` will activate all the loggers in that module and activating `PyExpLabSys` will activate all PyExpLabSys library loggers.

There are now two ways to activate a logger. One is to configure one from scratch, using the path of the logger and the same options as in `get_logger()`:

```
from PyExpLabSys.common import sockets
from PyExpLabSys.common.utilities import activate_library_logging

activate_library_logging(
    'PyExpLabSys.common.sockets.DateDataPullSocket',
    level='debug',
    file_log=True,
    file_name='socket_log.txt',
)
```

This would output all log message at debug level to a file called *socket_log.txt*.

The other way to activate a library logger is to ask it to enherit all the handlers and levels from an existing logger. This will send all the library log messages to the same destination:

```
from PyExpLabSys.common.utilities import get_logger, activate_library_logging
from PyExpLabSys.common import sockets

LOG = get_logger('my_program_name')
LOG.info('My program started')

# Configure a library logger to use the same handlers

activate_library_logging(
    'PyExpLabSys.common.sockets.DateDataPullSocket',
    logger_to_inherit_from=LOG,
)
```

It is still possible, when inheriting from an existing logger, to set a custom level for the library logger, using the `level` argument as in the example above.

2.3 Using PyExpLabSys drivers outside of PyExpLabSys

All I wanted was a banana, but what I got was a gorilla holding a banana

The quote above, is often used to refer to the fact that it can be difficult to use a component from a "framework" separate from the framework.

PyExpLabSys is not a framework as such, but there are some common elements that are used across different in principle independent modules. Specifically, most of the drivers in PyExpLabSys will work just fine outside of PyExpLabSys, with a few very minor modifications, by just copying the file to where the driver is to be used. Most of the drivers make use of just one other PyExpLabSys module that tie it to the package, the `supported_versions` module. The only thing that this module does, is to mark the specific driver as working with Python 2, Python 3 or both, and make a check at run-time of the Python version and possibly output a warning. It can therefore trivially be removed. To do this, look to lines of code somewhat like this:

```
from PyExpLabSys.common.supported_versions import python2_and_3
python2_and_3(__file__)
```

and comment them out. The specific function that is imported and called, will vary depending on which versions is supported, but that should be fairly simple to figure out.

CHAPTER 3

Common Software Components

This section documents the common software components at CINF. These are general purpose modules and classes that does not fit into the any of the other categories like `drivers` and `parsers`.

3.1 The database_saver module

3.1.1 Autogenerated API documentation for database_saver

Classes for saving coninuous data and data sets to a database

PyExpLabSys.common.database_saver.**HOSTNAME = u'servcinf-sql.fysik.dtu.dk'**
Hostname of the database server

PyExpLabSys.common.database_saver.**DATABASE = u'cinfdata'**
Database name

class PyExpLabSys.common.database_saver.**CustomColumn**(*value*, *format_string*)
Bases: `tuple`

format_string
Alias for field number 1

value
Alias for field number 0

class PyExpLabSys.common.database_saver.**DataSetSaver**(*measurements_table*, *xy_values_table*, *username*, *password*, *measurement_specs=None*)
Bases: `object`

A class to save a measurement

measurement_ids
Mapping of codenames to measurements ids

Type dict

measurements_table
The measurements tables

Type str

xy_values_table
The x, y values tables

Type str

sql_saver
The SqlSaver used to save points

Type *SqlSaver*

insert_measurement_query
The query used to insert a measurement

Type str

insert_point_query
The query used to insert a point

Type str

insert_batch_query
The query used to insert a batch of points

Type str

connection
The database connection used to register new measurements

Type MySQLdb connection

cursor
The database cursor used to register new measurements

Type MySQLdb cursor

__init__(*measurements_table*, *xy_values_table*, *username*, *password*, *measurement_specs=None*)
Initialize local parameters

Parameters

- **measurements_table** (*str*) – The name of the measurements table
- **xy_values_table** (*str*) – The name of the xy values table
- **username** (*str*) – The database username
- **passwork** (*str*) – The database password
- **measurement_specs** (*sequence*) – A sequence of `measurement_codename, metadata` pairs, see below

`measurement_specs` is used if you want to initialize all the measurements at `__init__` time. You can also do it later with *add_measurement()*. The expected value is a sequence of `measurement_codename, metadata` e.g:

```
measurement_specs = [
    ['M2', {'type': 5, 'timestep': 0.1, 'mass_label': 'M2M'}],
    ['M28', {'type': 5, 'timestep': 0.1, 'mass_label': 'M2M'}],
```

(continues on next page)

(continued from previous page)

```
    ['M32', {'type': 5, 'timestep': 0.1, 'mass_label': 'M2M'}],
]
```

As above, the expected `metadata` is simply a mapping of column names to column values in the `measurements_table`.

Per default, the value will be put into the table as is. If it is necessary to do SQL processing on the value, to make it fit the column type, then the value must be replaced with a *CustomColumn* instance, whose arguments are the value and the format/processing string. The format/processing string must contain a '%s' as a placeholder for the value. It could look like this:

```
measurement_specs = [
    ['M2', {'type': 5, 'time': CustomColumn(M2_timestamp, 'FROM_UNIXTIME(%s)
↪')}],
    ['M28', {'type': 5, 'time': CustomColumn(M28_timestamp, 'FROM_UNIXTIME(%s)
↪')}],
]
```

The most common use for this is the one shown above, where the `time` column is of type timestamp and the time value (e.g. in `M2_timestamp`) is a unix timestamp. The unix timestamp is converted to a SQL timestamp with the `FROM_UNIXTIME` SQL function.

Note: The SQL timestamp column understand the `datetime.datetime` type directly, so if the input timestamp is already on that form, then there is no need to convert it

---

add_measurement (*codename*, *metadata*)
Add a measurement

This is equivalent to forming the entry in the measurements table with the metadata values and saving the id of this entry locally for use with `add_point()`.

Parameters

- **codename** (*str*) – The codename that this measurement should have
- **metadata** (*dict*) – The dictionary that holds the information for the measurements table. See *__init__()* for details.

save_point (*codename*, *point*)
Save a point for a specific codename

Parameters

- **codename** (*str*) – The codename for the measurement to add the point to
- **point** (*sequence*) – A sequence of x, y

save_points_batch (*codename*, *x_values*, *y_values*, *batchsize=1000*)
Save a number points for the same codename in batches

Parameters

- **codename** (*str*) – The codename for the measurement to save the points for
- **x_values** (*sequence*) – A sequence of x values
- **y_values** (*sequence*) – A sequence of y values
- **batchsize** (*int*) – The number of points to send in the same batch. Defaults to 1000, see the warning below before changing it

> **Warning:** The batchsize is ultimately limited by the max package size that the MySQL server will receive. The default is 1MB. Each point amounts to around 60 bytes in the final query. Rounding this up to 100, means that the limit is ~10000 points. This means that the default of 1000 should be safe and that if it is changed by the user, expect problems if exceeding the lower 10000ths.

get_unique_values_from_measurements(*column*)
Return a set of unique column values from the measurements database

This is commonly used in fileparsers to identify the files already uploaded

Parameters **column** (*str*) – The column specification to extract values from. This can be just a column name e.g. "time", but it is also allowed to contain SQL processing e.g. UNIX_TIMESTAMP(time). The value of column will be formatted directly into the query.

start()
Start the DataSetSaver

And the underlying *SqlSaver*.

stop()
Stop the MeasurementSaver

And shut down the underlying *SqlSaver* instance nicely.

wait_for_queue_to_empty()
Wait for the query queue in the SqlSaver to empty

This purpose of this method is to avoid usgin too much memory when uploading large amount of data.

class PyExpLabSys.common.database_saver.**ContinuousDataSaver**(*continuous_data_table*, *username*, *password*, *measurement_codenames=None*)
Bases: object

This class saves data to the database for continuous measurements

Continuous measurements are measurements of a single parameters as a function of datetime. The class can ONLY be used with the new layout of tables for continous data, where there is only one table per setup, as apposed to the old layout where there was one table per measurement type per setup. The class sends data to the hostname and database named in *HOSTNAME* and *DATABASE* respectively.

__init__(*continuous_data_table*, *username*, *password*, *measurement_codenames=None*)
Initialize the continous logger

Parameters

- **continuous_data_table** (*str*) – The contunuous data table to log data to
- **username** (*str*) – The MySQL username
- **password** (*str*) – The password for username in the database
- **measurement_codenames** (*sequence*) – A sequence of measurement codenames that this logger will send data to. These codenames can be given here, to initialize them at the time of initialization or later by the use of the *add_continuous_measurement()* method.

> **Note:** The codenames are the 'official' codenames defined in the database for contionuous measurements NOT codenames that can be userdefined

add_continuous_measurement(*codename*)
Add a continuous measurement codename to this saver

Parameters **codename** (*str*) – Codename for the measurement to add

Note: The codenames are the 'official' codenames defined in the database for contionuous measurements NOT codenames that can be userdefined

save_point_now(*codename*, *value*)
Save a value and use now (a call to `time.time()`) as the timestamp

Parameters

- **codename** (*str*) – The measurement codename that this point will be saved under
- **value** (*float*) – The value to be logged

Returns The Unixtime used

Return type float

save_point(*codename*, *point*)
Save a point

Parameters

- **codename** (*str*) – The measurement codename that this point will be saved under
- **point** (*sequence*) – The point to be saved, as a sequence of 2 floats: (x, y)

start()
Starts the underlying *SqlSaver*

stop()
Stop the ContiniousDataSaver

And shut down the underlying *SqlSaver* instance nicely.

class PyExpLabSys.common.database_saver.**SqlSaver**(*username*, *password*, *queue=None*)
Bases: threading.Thread

The SqlSaver class administers a queue from which it executes SQL queries

Note: In general queries are added to the queue via the *enqueue_query()* method. If it is desired to add elements manually, remember that they must be on the form of a `(query, query_args)` tuple. (These are the arguments to the execute method on the cursor object)

queue
The queue the queries and qeury arguments are stored in. See note below.

Type Queue.Queue

commits
The number of commits the saver has performed

Type int

commit_time
The timespan the last commit took

Type float

connection
The MySQLdb database connection

Type MySQLdb connection

cursor
The MySQLdb database cursor

Type MySQLdb cursor

__init__(*username*, *password*, *queue=None*)
Initialize local variables

Parameters

- **username** (*str*) – The username for the MySQL database
- **password** (*str*) – The password for the MySQL database
- **queue** (*Queue.Queue*) – A custom queue to use. If it is left out, a new Queue.Queue object will be used.

stop()
Add stop word to queue to exit the loop when the queue is empty

enqueue_query(*query*, *query_args=None*)
Enqueue a qeury and arguments

Parameters

- **query** (*str*) – The SQL query to be executed
- **query_args** (*sequence or mapping*) – Optional sequence or mapping of arguments to be formatted into the query. query and query_args in combination are the arguments to cursor.execute.

run()
Execute SQL inserts from the queue until stopped

wait_for_queue_to_empty()
Wait for the queue to empty

This purpose of this method is to avoid using too much memory when uploading large amount of data.

PyExpLabSys.common.database_saver.**run_module**()
Run the module to perform elementary functional test

3.2 The loggers module

The logger module contains classes that hide some of all the repeated code associated with sending data to the data base. The main component is the ContinousLogger class, which is used to send continuously logged data to the data base.

3.2.1 The continuous logger

The *ContinuousLogger* class is meant to do as much of the manual work related to logging a parameter continuously to the database as possible. The main features are:

- **Simple single method usage.** After the class is instantiated, a single call to *enqueue_point()* or *enqueue_point_now()* is all that is needed to log a point. No manual database query manipulation is required.
- **Resistent to network or database down time.** The class implements a queue for the data, from which points will only be removed if they are successfully handed of to the data base and while it is not possible to hand the data of, it will be stored in memory.

Warning: The resilience against network downtime has only been tested for the way it will fail if you disable the network from the machine it is running on. Different kinds of failures may produce different kinds of failure modes. If you encounter a failure mode that the class did not recover from you should report it as a bug.

Todo: Write that it uses new style database layout and refer to that section.

Usage Example

If code already exists to retrieve the data (e.g. a driver to interface a piece of equipment with), writing a data logger can be reduced to as little as the following:

```
from PyExpLabSys.common.loggers import ContinuousLogger

db_logger = ContinuousLogger(table='dateplots_dummy',
                             username='dummy', password='dummy',
                             measurement_codenames = ['dummy_sine_one'])
db_logger.start()

# Initialize variable for the logging condition
while True:
    new_value = driver.get_value()
    if contition_to_log_is_true:
        db_logger.enqueue_point_now('dummy_sine_one', new_value)
```

or if it is preferred to keep track of the timestamp manually:

```
import time
from PyExpLabSys.common.loggers import ContinuousLogger

# Initiate the logger to write to the dateplots_dummy table, with usernam
db_logger = ContinuousLogger(table='dateplots_dummy',
                             username='dummy', password='dummy',
                             measurement_codenames = ['dummy_sine_one'])
db_logger.start()

# Initialize variable for the logging condition
while True:
    new_value = driver.get_value()
    now = time.time()
    if contition_to_log_is_true:
        db_logger.enqueue_point('dummy_sine_one', (now, new_value))
```

Auto-generated module documentation

This module contains convinience classes for database logging. The classes implement queues to contain the data before of loading to the database to ensure against network or server problems.

class PyExpLabSys.common.loggers.**NoneResponse**

__init__()

PyExpLabSys.common.loggers.**NONE_RESPONSE = <PyExpLabSys.common.loggers.NoneResponse instanc**
Module variable used to indicate a none response, currently is an instance if NoneResponse

class PyExpLabSys.common.loggers.**InterruptableThread**(*cursor*, *query*)
Bases: threading.Thread

Class to run a MySQL query with a time out

__init__(*cursor*, *query*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Start the thread

PyExpLabSys.common.loggers.**timeout_query**(*cursor*, *query*, *timeout_duration=3*)
Run a mysql query with a timeout

Parameters

- **cursor** (*MySQL cursor*) – The database cursor
- **query** (*str*) – The query to execute
- **timeout_duration** (*int*) – The timeout duration

Returns

(tuple or *NONE_RESPONSE*): A tuple of results from the query or *NONE_RESPONSE* if the query timed out

exception PyExpLabSys.common.loggers.**StartupException**(**args*, ***kwargs*)
Bases: exceptions.Exception

Exception raised when the continous logger fails to start up

__init__(**args*, ***kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

class PyExpLabSys.common.loggers.**ContinuousLogger**(*table*, *username*, *password*, *measurement_codenames*, *dequeue_timeout=1*, *reconnect_waittime=60*, *dsn=None*)

Bases: threading.Thread

A logger for continous data as a function of datetime. The class can ONLY be used with the new layout of tables for continous data, where there is only one table per setup, as apposed to the old layout where there was one table per measurement type per setup. The class sends data to the cinfdata database at host servcinf-sql.

Variables

- **host** – Database host, value is servcinf-sql.
- **database** – Database name, value is cinfdata.

__init__(*table*, *username*, *password*, *measurement_codenames*, *dequeue_timeout=1*, *reconnect_waittime=60*, *dsn=None*)

Initialize the continous logger

Parameters

- **table** (*str*) – The table to log data to
- **username** (*str*) – The MySQL username (must have write rights to table)
- **password** (*str*) – The password for user in the database
- **measurement_codenames** (*list*) – List of measurement codenames that this logger will send data to
- **dequeue_timeout** (*float*) – The timeout (in seconds) for dequeueing an element, which also constitutes the max time to shutdown after the thread has been asked to stop. Default is 1.
- **reconnect_waittime** (*float*) – Time to wait (in seconds) in between attempts to re-connect to the MySQL database, if the connection has been lost
- **dsn** (*str*) – DSN name of ODBC connection, used on Windows only

Raises ***StartupException*** – If it is not possible to start the database connection or translate the code names

stop()

Stop the thread

run()

Start the thread. Must be run before points are added.

enqueue_point_now(*codename*, *value*)

Add a point to the queue and use the current time as the time

Parameters

- **codename** (*str*) – The measurement codename that this point will be saved under
- **value** (*float*) – The value to be logged

Returns The Unixtime used

Return type float

enqueue_point(*codename*, *point*)

Add a point to the queue

Parameters

- **codename** (*str*) – The measurement codename that this point will be saved under
- **point** (*iterable*) – Current point as a list (or tuple) of 2 floats: [x, y]

3.3 The plotters module plotter backends

The plotters module contains classes for easy plotting of data.

3.3.1 The data logger

The `DataLogger` is a general purpose plotter that is suitable for plotting data sets as they are being gathered. The data logger uses the qwt backend, though the class *QwtPlot*, that forms the plot by means of the PyQwt library.

Usage Example

```
import sys
import time
import random
import numpy as np
from PyQt4 import Qt, QtGui, QtCore
from PyExpLabSys.common.plotters import DataPlotter

class TestApp(QtGui.QWidget):
    """Test Qt application"""

    def __init__(self):
        super(TestApp, self).__init__()
        # Form plotnames
        self.plots_l = ['signal1', 'signal2']
        self.plots_r = ['aux_signal1']

        self.plotter = DataPlotter(
            self.plots_l, right_plotlist=self.plots_r, parent=self,
            left_log=True, title='Awesome plots',
            yaxis_left_label='Log sine, cos', yaxis_right_label='Noisy line',
            xaxis_label='Time since start [s]',
            legend='right', left_thickness=[2, 8], right_thickness=6,
            left_colors=['firebrick', 'darkolivegreen'],
            right_colors=['darksalmon'])

        hbox = QtGui.QHBoxLayout()
        hbox.addWidget(self.plotter.plot)
        self.setLayout(hbox)
        self.setGeometry(5, 5, 500, 500)

        self.start = time.time()
        QtCore.QTimer.singleShot(10, self.main)

    def main(self):
        """Simulate gathering one set of points and adding them to plot"""
        elapsed = time.time() - self.start
        value = (np.sin(elapsed) + 1.1) * 1E-9
        self.plotter.add_point('signal1', (elapsed, value))
```

(continues on next page)

(continued from previous page)

```
        value = (np.cos(elapsed) + 1.1) * 1E-8
        self.plotter.add_point('signal2', (elapsed, value))
        value = elapsed + random.random()
        self.plotter.add_point('aux_signal1', (elapsed, value))

        QtCore.QTimer.singleShot(100, self.main)

def main():
    """Main method"""
    app = Qt.QApplication(sys.argv)
    testapp = TestApp()
    testapp.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

plotters module

This module contains plotters for experimental data gathering applications. It contains a plotter for data sets.

class PyExpLabSys.common.plotters.**DataPlotter**(*left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, *auto_update=True*, *backend='qwt'*, *parent=None*, ***kwargs*)

Bases: object

This class provides a data plotter for continuous data

__init__(*left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, *auto_update=True*, *backend='qwt'*, *parent=None*, ***kwargs*)

Initialize the plotting backend, data and local setting

Parameters

- **left_plotlist** (*iterable with strs*) – Codenames for the plots that should go on the left y-axis
- **right_plotlist** – Codenames for the plots that should go in the right y-axis
- **left_log** (*bool*) – Left y-axis should be log
- **right_log** (*bool*) – Right y-axis should be log
- **auto_update** (*bool*) – Whether all data actions should trigger an update
- **backend** (*str*) – The plotting backend to use. Current only option is 'qwt'
- **parent** (*GUI object*) – If a GUI backend is used that needs to know the parent GUI object, then that should be supplied here

Kwargs:

Parameters

- **title** (*str*) – The title of the plot
- **xaxis_label** (*str*) – Label for the x axis
- **yaxis_left_label** (*str*) – Label for the left y axis
- **yaxis_right_label** (*str*) – Label for the right y axis

- **left_labels** (*iterable with strs*) – Labels for the plots on the left y-axis. If none are given the codenames will be used.
- **right_labels** (*iterable with strs*) – Labels for the plots on the right y-axis. If none are given the codenames will be used.
- **legend** (*str*) – Position of the legend. Possible values are: 'left', 'right', 'bottom', 'top'. If no argument is given, the legend will not be shown.
- **left_colors** (*iterable of strs*) – Colors for the left curves (see background_color for details)
- **right_colors** (*iterable of strs*) – Colors for the right curves (see background_color for details)
- **left_thickness** (*int or iterable of ints*) – Line thickness. Either an integer to apply for all left lines or a iterable of integers, one for each line.
- **right_thickness** (*int or iterable of ints*) – Line thickness. Either an integer to apply for all right lines or a iterable of integers, one for each line.
- **background_color** (*str*) – The name in a str (as understood by QtGui.QColor(), see *Colors* section for possible values) or a string with a hex value e.g. '#101010' that should be used as the background color.

add_point(*plot*, *point*, *update=None*)
Add a point to a plot

Parameters

- **plot** (*str*) – The codename for the plot
- **point** (*Iterable with x and y value as two numpy.float*) – The point to add
- **update** – Whether a update should be performed after adding the point. If set, this value will over write the `auto_update` value

Returns plot content or None

update()
Update the plot and possible return the content

data
Get and set the data

plot
Get the plot

class PyExpLabSys.common.plotters.**ContinuousPlotter**(*left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, *timespan=600*, *preload=60*, *auto_update=True*, *backend='none'*, ***kwargs*)

Bases: object

This class provides a data plotter for continuous data

__init__(*left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, *timespan=600*, *preload=60*, *auto_update=True*, *backend='none'*, ***kwargs*)
Initialize the plotting backend, data and local setting

Parameters

- **left_plotlist** (*iterable with strs*) – Codenames for the plots that should go on the left y-axis
- **right_plotlist** – Codenames for the plots that should go in the right y-axis
- **left_log** (*bool*) – Left y-axis should be log
- **right_log** (*bool*) – Right y-axis should be log
- **timespan** (*int*) – Numbers of seconds to show in the plot
- **preload** (*int*) – Number of seconds to jump ahead when reaching edge of plot
- **auto_update** (*bool*) – Whether all data actions should trigger a update
- **backend** (*str*) – The plotting backend to use. Current only option is 'none'

Kwargs

TODO

add_point_now(*plot*, *value*, *update=None*)
Add a point to a plot using now as the time

Parameters

- **plot** (*str*) – The codename for the plot
- **value** (*numpy.float*) – The value to add
- **update** – Whether a update should be performed after adding the point. If set, this value will over write the `auto_update` value

Returns plot content or None

add_point(*plot*, *point*, *update=None*)
Add a point to a plot

Parameters

- **plot** (*str*) – The codename for the plot
- **point** (*Iterable with unix time and value as two numpy.float*) – The point to add
- **update** – Whether a update should be performed after adding the point. If set, this value will over write the `auto_update` value

Returns plot content or None

update()
Update the plot and possible return the content

data
Get and set the data

plot
Get the plot

3.3.2 The qwt backend

plotters_backend_qwt

This module contains plotting backends that use the PyQwt library

class PyExpLabSys.common.plotters_backend_qwt.**Colors**
Class that gives plot colors

__init__()

get_color()
Return a color

class PyExpLabSys.common.plotters_backend_qwt.**QwtPlot**(*parent*, *left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, ***kwargs*)
Bases: list

Class that represents a Qwt plot

__init__(*parent*, *left_plotlist*, *right_plotlist=None*, *left_log=False*, *right_log=False*, ***kwargs*)
Initialize the plot and local setting

Parameters

- **parent** (*GUI object*) – The parent GUI object, then that should be supplied here
- **left_plotlist** (*iterable with strs*) – Codenames for the plots that should go on the left y-axis
- **right_plotlist** – Codenames for the plots that should go in the right y-axis
- **left_log** (*bool*) – Left y-axis should be log
- **right_log** (*bool*) – Right y-axis should be log

Kwargs:

Parameters

- **title** (*str*) – The title of the plot
- **xaxis_label** (*str*) – Label for the x axis
- **yaxis_left_label** (*str*) – Label for the left y axis
- **yaxis_right_label** (*str*) – Label for the right y axis
- **left_labels** (*iterable with strs*) – Labels for the plots on the left y-axis. If none are given the codenames will be used.
- **right_labels** (*iterable with strs*) – Labels for the plots on the right y-axis. If none are given the codenames will be used.
- **legend** (*str*) – Position of the legend. Possible values are: 'left', 'right', 'bottom', 'top'. If no argument is given, the legend will not be shown.
- **left_colors** (*iterable of strs*) – Colors for the left curves (see background_color for details)
- **right_colors** (*iterable of strs*) – Colors for the right curves (see background_color for details)
- **left_thickness** (*int or iterable of ints*) – Line thickness. Either an integer to apply for all left lines or a iterable of integers, one for each line.
- **right_thickness** (*int or iterable of ints*) – Line thickness. Either an integer to apply for all right lines or a iterable of integers, one for each line.

- **background_color** (*str*) – The name in a str (as understood by QtGui.QColor(), see *Colors* section for possible values) or a string with a hex value e.g. '#101010' that should be used as the background color.

update(*data*)

Update the plot with new values and possibly move the xaxis

Parameters **data** (*dict*) – The data to plot. Should be a dict, where keys are plot code names and values are data series as an iterable of (x, y) iterables. E.g. {'plot1': [(1, 1), (2, 2)]}

Colors

aliceblue	antiquewhite	aqua	aquamarine
azure	beige	bisque	black
blanchedalmond	blue	blueviolet	brown
burlywood	cadetblue	chartreuse	chocolate
coral	cornflowerblue	cornsilk	crimson
cyan	darkblue	darkcyan	darkgoldenrod
darkgray	darkgreen	darkgrey	darkkhaki
darkmagenta	darkolivegreen	darkorange	darkorchid
darkred	darksalmon	darkseagreen	darkslateblue
darkslategray	darkslategrey	darkturquoise	darkviolet
deeppink	deepskyblue	dimgray	dimgrey
dodgerblue	firebrick	floralwhite	forestgreen
fuchsia	gainsboro	ghostwhite	gold
goldenrod	gray	green	greenyellow
grey	honeydew	hotpink	indianred
indigo	ivory	khaki	lavender
lavenderblush	lawngreen	lemonchiffon	lightblue
lightcoral	lightcyan	lightgoldenrodyellow	lightgray
lightgreen	lightgrey	lightpink	lightsalmon
lightseagreen	lightskyblue	lightslategray	lightslategrey
lightsteelblue	lightyellow	lime	limegreen
linen	magenta	maroon	mediumaquamarine
mediumblue	mediumorchid	mediumpurple	mediumseagreen
mediumslateblue	mediumspringgreen	mediumturquoise	mediumvioletred
midnightblue	mintcream	mistyrose	moccasin
navajowhite	navy	oldlace	olive
olivedrab	orange	orangered	orchid
palegoldenrod	palegreen	paleturquoise	palevioletred
papayawhip	peachpuff	peru	pink
plum	powderblue	purple	red
rosybrown	royalblue	saddlebrown	salmon
sandybrown	seagreen	seashell	sienna
silver	skyblue	slateblue	slategray
slategrey	snow	springgreen	steelblue
tan	teal	thistle	tomato
transparent	turquoise	violet	wheat
white	whitesmoke	yellow	yellowgreen

3.4 The sockets module

The data sockets are convenience classes that make it easier to send data back and forth between machines. All the data sockets are socket **servers**, i.e. they handle requests, and to interact with them it is necessary to work as a client. The main purpose of these sockets is to hide the complexity and present a easy-to-use interface while performing e.g. error checking in the background.

The sockets are divided into push and pull sockets, which are intended to either pull data from or pushing data to.

The main advantages of the **pull sockets** are:

- **Simple usage**: After e.g. the `DateDataPullSocket` or `DataPullSocket` class is instantiated, a single call to the `set_point` method is all that is needed to make a point available via the socket.
- **Codename based**: After instantiation, the different data slots are referred to by codenames. This makes code easier to read and help to prevent e.g. indexing errors.
- **Timeout safety to prevent serving obsolete data**: The class can be instantiated with a timeout for each measurement type. If the available point is too old an error message will be served.

The main advantages of the **push socket** are:

- **Simple usage**: If all that is required is to receive data in a variable like manner, both the last and the updated variable values can be accessed via the `DataPushSocket.last` and `DataPushSocket.updated` properties.
- **Flexible**: The `DataPushSocket` offers a lot of functionality around what actions are performed when a data set is received, including enqueue it or calling a callback function with the data set as an argument.

Table of Contents

3.4.1 Examples

In the following examples it is assumed, that all other code that is needed, such as e.g. an equipment driver, already exists, and the places where such code is needed, is filled in with dummy code.

DateDataPullSocket make data available (network variable)

Making data available on the network for pulling can be achieved with:

```
from PyExpLabSys.common.sockets import DateDataPullSocket

# Create a data socket with timeouts and start the socket server
name = 'Last shot usage data from the giant laser on the moon'
codenames = ['moon_laser_power', 'moon_laser_duration']
moon_socket = DateDataPullSocket(name, codenames, timeouts=[1.0, 0.7])
moon_socket.start()

try:
    while True:
        power, duration = laser.get_values()
        # To set a variable use its codename
        moon_socket.set_point_now('moon_laser_power', power)
        moon_socket.set_point_now('moon_laser_duration', duration)
except KeyboardInterrupt:
    # Stop the socket server
    moon_socket.stop()
```

or if it is preferred to keep track of the timestamp manually:

```
try:
    while True:
        now = time.time()
        power, duration = driver.get_values()
        moon_socket.set_point('moon_laser_power', (now, power))
        moon_socket.set_point('moon_laser_duration', (now, duration))
except KeyboardInterrupt:
    # Stop the socket server
    moon_socket.stop()
```

A few things to note from the examples. The port number used for the socket is 9000, which is the default for this type is socket (see *Port defaults*). The two measurements have been setup to have different timeouts (maximum ages), which is in seconds by the way, but it could also be set up to be the same, and if it is the same, it can be supplied as just one float `timeouts=0.7` instead of a list of floats. For the sockets, the codenames should be kept relatively short, but for data safety reasons, they should contain an unambiguous reference to the setup, i.e. the 'moon_laser' part.

Client side

The client can be set up in the following manner:

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# hostname can something like 'rasppi250', if that happens to be the one
# located on the moon
host_port = ('moon_raspberry_pi', 9000)
```

Command and data examples

With the initialization of the client as above, it is now possible to send the socket different commands and get appropriate responses. In the following, the different commands are listed, along with how to send it, receive (and decode) the reply.

The `name` command

Used to get the name of the socket server, which can be used to make sure that data is being pulled from the correct location:

```
command = 'name'
sock.sendto(command, host_port)
data = sock.recv(2048)
```

at which point the data variable, which contains the reply, will contain the string `'Last shot usage data from the giant laser on the moon'`.

The `json_wn` command

Tip: The 'wn' suffix is short for 'with names' and is used for all the sockets to indicate that data is sent or received prefixed with names of the particular data channel

This command is used to get all the latest data encoded as `json`. It will retrieve all the data as a dictionary where the keys are the names, encoded as `json` for transport:

```
import json
command = 'json_wn'
sock.sendto(command, host_port)
data = json.loads(sock.recv(2048))
```

at which point the data variable will contain a dict like this one:

```
{u'moon_laser_power': [1414150015.697648, 47.0], u'moon_laser_duration': [1414150015.
↪697672, 42.0]}
```

The `codenames_json` and `json` commands

It is also possible to decouple the codenames. A possible use case might be to produce a plot of the data. In such a case, the names are really only needed when setting up the plot, and then afterwards the data should just arrive in the same order, to add points to the graph. This is exactly what these two command do:

```
import json

# Sent only once
command = 'codenames_json'
sock.sendto(command, host_port)
codenames = json.loads(sock.recv(2048))

# Sent repeatedly
command = 'json'
sock.sendto(command, host_port)
data = json.loads(sock.recv(2048))
```

after which the codenames variable would contain a list of codenames:

```
[u'moon_laser_power', u'moon_laser_duration']
```

and the data variable would contain a list of points, returned in the same order as the codenames:

```
[[1414150538.551638, 47.0], [1414150538.551662, 42.0]]
```

The `codename#json` command

Note: The codename in the command should be substituted with an actual codename

It is also possible to ask for a single point by name.:

```
import json
command = 'moon_laser_power#json'
sock.sendto(command, host_port)
data = json.loads(sock.recv(2048))
```

At which point the data variable would contains just a single point as a list:

```
[1414151366.400581, 47.0]
```

The `raw_wn`, `codenames_raw`, `raw` and `codename#raw` commands

These commands do exactly the same as their `json` counterparts, only that the data is not encoded as `json`, but with the homemade raw encoding.

Warning: The raw encoding is manually serialized, which is an 100% guarantied error prone approach, so use the `json` variant whenever possible. Even Labview® to some extent supports json as of version 2013.

Remember that when receiving data in the raw encoding, it should not be json decoded, so the code to work with these commands will look like this:

```
command = 'some_raw_command'  # E.g. raw_wn
sock.sendto(command, host_port)
data = sock.recv(2048)
```

There format if the raw encoding is documented in the API documentation for the *`PullUDPHandler.handle()`* method.

Below are a simple list of each type of raw command and example out:

```
command = 'raw_wn'
# Output
'moon_laser_power:1414154338.17,47.0;moon_laser_duration:1414154338.17,42.0'

command = 'codenames_raw'
# Output
'moon_laser_power,moon_laser_duration'

command = 'raw'
# Output
'1414154433.4,47.0;1414154433.4,42.0'
```

(continues on next page)

(continued from previous page)

```
command = 'moon_laser_power#raw'
# Output
'1414154485.08,47.0'
```

DataPushSocket, send data (network variable)

To receive data on a machine, the *DataPushSocket* can be used. To set it up simply to be able to see the last received data and the updated total data, set it up like this:

```
from PyExpLabSys.common.sockets import DataPushSocket
name = 'Data receive socket for giant laser on the moon'
dps = DataPushSocket(name, action='store_last')
# Get data
timestamp_last, last = dps.last
timestamp_updated, updated = dps.updated
# ... do whatever and stop socket
dps.stop()
```

After settings this up, the last received data set will be available as a tuple in the *DataPushSocket.last* property, where the first value is the Unix timestamp of reception and the second value is a dictionary with the last received data (all data sent to the *DataPushSocket* is in dictionary form, see *PushUDPHandler.handle()* for details). Alternatively, the *DataPushSocket.updated* property contains the last value received ever for each key in the dict (i.e. not only in the last transmission).

Command examples

The socket server understands commands in two formats, a `json` and a raw encoded one. For details about how to send commands to a socket server and receive the reply in the two different encodings, see the sections *Client side* and *Command and data examples* from the *DateDataPullSocket make data available (network variable)* section.

The `json` commands looks like this:

```
json_wn#{"greeting": "Live long and prosper", "number": 47}
json_wn#{"number": 42}
```

After the first command the data values in both the *DataPushSocket.last* and the *DataPushSocket.updated* properties are:

```
{u'greeting': u'Live long and prosper', u'number': 47}
```

After the second command the value in the *DataPushSocket.last* property is:

```
{u'number': 42}
```

and the value in the *DataPushSocket.updated* property is:

```
{u'greeting': u'Live long and prosper', u'number': 42}
```

The commands can also be raw encoded, in which case the commands above will be:

```
raw_wn#greeting:str:Live long and prosper;number:int:47
raw_wn#number:int:42
```

Warning: See the *warning about the raw encoding*.

Upon receipt, the socket server will make a message available on the socket, that contains a status for the receipt and a copy of the data it has gathered (in simple Python string representation). I will look like this:

```
ACK#{u'greeting': u'Live long and prosper', u'number': 47}
```

If it does not understand the data, e.g. if it is handed badly formatted raw data, it will return an error:

```
Sending: "raw_wn#number:88"
Will return: "ERROR#The data part 'number:88' did not match the expected format of 3␣
↪parts divided by ':'"

Sending: "raw_wn#number:floats:88"
Will return: "ERROR#The data type 'floats' is unknown. Only ['int', 'float', 'bool',
↪'str'] are allowed"
```

DataPushSocket, see all data sets received (enqueue them)

To receive data and make sure that each and every point is reacted to, it is possible to ask the socket to enqueue the data. It is set up in the following manner:

```
from PyExpLabSys.common.sockets import DataPushSocket
name = 'Command receive socket for giant laser on the moon'
dps = DataPushSocket(name, action='enqueue')
queue = dps.queue  # Local variable to hold the queue
# Get on point
print queue.get()
# Get all data
for _ in range(queue.qsize()):
    print queue.get()
# ... do whatever and stop socket
dps.stop()
```

As seen above, the queue that holds the data items, is available as the *DataPushSocket.queue* attribute. Data can be pulled out by calling `get()` on the queue. NOTE: The for-loop only gets all the data that was available at the time of calling `qsize()`, so if the actions inside the for loop takes time, it is possible that new data will be enqueued while the for-loop is running, which it will not pull out.

If it is desired to use an existing queue or to set up a queue with other than default settings, the *DataPushSocket* can be instantiated with a custom queue.

Command examples

Examples of commands that can be sent are the same as in the *code example above*, after which the queue would end up containing the two dictionaries:

```
{u'greeting': u'Live long and prosper', u'number': 47}
{u'number': 42}
```

DataPushSocket, make socket call function on reception (callback)

With the *DataPushSocket* it is also possible to ask the socket to call a callback function on data reception:

```
from PyExpLabSys.common.sockets import DataPushSocket
import time

# Module variables
STATE = 'idle'
STOP = False

def callback_func(data):
    """Callback function for the socket"""
    print 'Received: {}'.format(data)
    #... do fancy stuff depending on the data, e.g. adjust laser settings
    # or fire (may change STATE)

name = 'Command callback socket for giant laser on the moon'
dps = DataPushSocket(name, action='callback_async', callback=callback_func)

while not STOP:
    # Check if there is a need for continuous activity e.g. monitor
    # temperature of giant laser during usage
    if STATE == 'fire':
        # This function should end when STATE changes away from 'fire'
        monitor_temperature()
    time.sleep(1)

# After we are all done, stop the socket
dps.stop()
```

In this examples, the data for the callbacks (and therefore the callbacks themselves) will be queued up and happen asynchronously. This makes it possible to send a batch of commands without waiting, but there is no monitoring of whether the queue is filled faster than it can be emptied. It can of course be checked by the user, but if there is a need for functionality in which the sockets make such checks itself and rejects data if there is too much in queue, then talk to the development team about it.

Command examples

Command examples this kind of socket will as always be dicts, but in this case will likely have to contain some information about which action to perform or method to call, but that is entirely up to the user, since that is implemented by the user in the call back function. Some examples could be:

```
json_wn#{"action": "fire", "duration": 8, "intensity": 300}
json_wn#{"method": "fire", "duration": 8, "intensity": 300}
```

DataPushSocket, control class and send return values back (callback with return)

This is reduced version of an example that shows two things:

- How to get the return value when calling a function via the *DataPushSocket*
- How to control an entire class with a *DataPushSocket*

```
from PyExpLabSys.common.sockets import DataPushSocket

class LaserControl(object):
    """Class that controls the giant laser laser on the moon"""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self.settings = {'power': 100, 'focus': 100, 'target': None}
        self._state = 'idle'

        # Start socket
        name = 'Laser control, callback socket, for giant laser on the moon'
        self.dps = DataPushSocket(name, action='callback_direct',
                                  callback=self.callback)
        self.dps.start()

        self.stop = False
        # Assume one of the methods can set stop
        while not self.stop:
            # Do continuous stuff on command
            time.sleep(1)
        self.dps.stop()

    def callback(self, data):
        """Callback and central control function"""
        # Get the method name and don't pass it on as an argument
        method_name = data.pop('method')
        # Get the method
        method = self.__getattribute__(method_name)
        # Call the method and return its return value
        return method(**data)

    def update_settings(self, **kwargs):
        """Update settings"""
        for key in kwargs.keys():
            if key not in self.settings.keys():
                message = 'Not settings for key: {}'.format(key)
                raise ValueError(message)
        self.settings.update(kwargs)
        return 'Updated settings with: {}'.format(kwargs)

    def state(self, state):
        """Set state"""
        self._state = state
        return 'State set to: {}'.format(state)
```

This socket would then be sent commands in the form of `json` encoded dicts from an UDP client in the secret lair. These dicts could look like:

```
{'method': 'update_settings', 'power': 300, 'focus': 10}
# or
{'method': 'state', 'state': 'active'}
```

which would, respectively, make the `update_settings` method be called with the arguments `power=300`, `focus=10` and the `state` method be called with the argument `state='active'`. NOTE: In this implementation, it is the responsibility of the caller that the method name exists and that the arguments that are sent have the correct names. An alternative, but less flexible, way to do the same, would be to make an if-elif-else structure on the method name and format the arguments in manually:

```
def callback(self, data):
    """Callback and central control function"""
```

(continues on next page)

(continued from previous page)

```
    method_name = data.get('method')
    if method_name == 'state':
        if data.get('state') is None:
            raise ValueError('Argument \'state\' missing')
        out = self.state(data['state'])
    elif method_name == 'update_settings':
        # ....
        pass
    else:
        raise ValueError('Unknown method: {}'.format(method_name))

    return out
```

The complete and running example of both server and client for this example can be downloaded in these two files: `server`, `client`.

Command examples

See the attached files with example code for command examples.

3.4.2 Port defaults

To make for easier configuration on both ends of the network communication, the different kinds of sockets each have their own default port. They are as follows:

Socket	Default port
DateDataPullSocket	9000
DataPullSocket	9010
DataPushSocket	8500
LiveSocket	8000

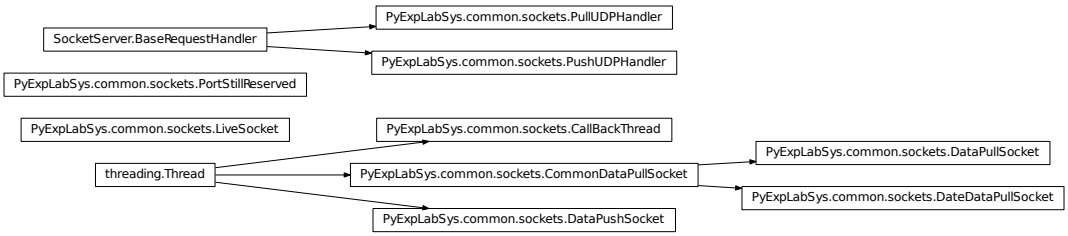
Again, to ease configuration also on the client side, if more than one socket of the same kind is needed on one machine, then it is recommended to simply add 1 to the port number for each additional socket.

3.4.3 Inheritance

The `DateDataPullSocket` and `DataPullSocket` classes inherit common functionality, such as;

- Input checks
- Initialization of DATA
- Methods to start and stop the thread and reset DATA

from the `CommonDataPullSocket` class, as illustrated in the diagram below.



3.4.4 Status of a socket server

All 4 socket servers understand the `status` command. This command will return some information about the status of the machine the socket server is running on and the status of **all** socket servers running on this machine. The reason the command returns the status for all the socket servers running on the machine is, that what this command is really meant for, is to get the status of *the system* and so it should not be necessary to communicate with several socket servers to get that.

The data returned is a `json` encoded dictionary, which looks like this:

```
{u'socket_server_status':
          {u'8000': {u'name': u'my_live_socket',
                     u'since_last_activity': 0.0009279251098632812,
                     u'status': u'OK',
                     u'type': u'live'},
           u'9000': {u'name': u'my_socket',
                     u'since_last_activity': 0.0011229515075683594,
                     u'status': u'OK',
                     u'type': u'date'}},
 u'system_status':
          {u'filesystem_usage': {u'free_bytes': 279182213120,
                                 u'total_bytes': 309502345216},
           u'last_apt_cache_change_unixtime': 1413984912.529932,
           u'last_git_fetch_unixtime': 1413978995.4109764,
           u'load_average': {u'15m': 0.14, u'1m': 0.1, u'5m': 0.15},
           u'max_python_mem_usage_bytes': 37552128,
           u'number_of_python_threads': 3,
           u'uptime': {u'idle_sec': 321665.77,
                       u'uptime_sec': 190733.39}}}
```

Socket server status

The **socket servers status** is broken down into one dictionary for each socket server, indexed by their ports. The status for the individual socket server comprise of the following items:

name (*str*) The name of the socket server

since_last_activity (*float*) The number of seconds since last activity on the socket server

status (*str*) The status of the socket server. It will return either `'OK'` if the last activity was newer than the activity timeout, or `'INACTIVE'` if the last activity was older than the activity timeout or `'DISABLED'` is activity monitoring is disabled for the socket server.

type The type of the socket server

System status

The **system status** items depends on the operating system the socket server is running on.

All operating systems

last_git_fetch_unixtime (*float*) The Unix timestamp of the last git fetch of the 'origin' remote, which points at the Github archive

max_python_mem_usage_bytes (*int*) The maximum memory usage of Python in bytes

number_of_python_threads (*int*) The number of Python threads in use

Linux operating systems

uptime (*dict*) Information about system uptime (from `/proc/uptime`). The value `'uptime_sec'` contains the system uptime in seconds and the value `'idle_sec'` contains the idle time in seconds. NOTE: While the uptime is measured in wall time, the idle time is measured in CPU time, which means that if the system is multi-core, it will add up idle time for all the cores.

last_apt_cache_change_unixtime (*float*) The Unix time stamp of the last change to the apt cache, which should be a fair approximation to the last time the system was updated

load_average(*dict*) The load average (roughly number of active processes) over the last 1, 5 and 15 minutes (from `/proc/loadavg`). For a detailed explanation see the /proc/loadavg section from the proc man-page

filesystem_usage (*dict*) The number of total and free bytes for the file-system the PyExpLabSys archive is located on

3.4.5 Auto-generated module documentation

The sockets module contains various implementations of UDP socket servers (at present 4 in total) for transmission of data over the network. The different implementations are tailored for a specific purposes, as described below.

In general, there is a distinction in the naming of the different socket server implementation between **push** socket servers, that you can push data to, and **pull** socket servers, that you can pull data from.

Presently the module contains the following socket servers:

- **DateDataPullSocket** (*DateDataPullSocket*) This socket server is used to make continuous data (i.e. one-value data as function of time) available on the network. It identifies different data channels by codenames and has a timeout functionality to prevent serving old data to the client.
- **DataPullSocket** (*DataPullSocket*) This socket is similar to the date data server, but is used to make x, y type data available on the network. It identifies different data channel by codenames and has timeout functionality to prevent serving old data.
- **DataPushSocket** (*DataPushSocket*) This socket is used to recieve data from the network. The data is received in dictionary form and it identifies the data channels by codenames (keys) in the dictionaries. It will save the last point, and the last value for each codename. It also has the option to, for each received data set, to put them in a queue (that the user can then empty) or to call a callback function with the received data as en argument.

- **LiveSocket** (*LiveSocket*) This socket is used only for serving data to the live socket server. It also is not actually a socket server like the others, but it has a similar interface.

Note: The module variable *DATA* is a dict shared for all socket servers started from this module. It contains all the data, queues, settings etc. It can be a good place to look if, to get a behind the scenes look at what is happening.

PyExpLabSys.common.sockets.**bool_translate**(*string*)
Returns boolean value from strings 'True' or 'False'

PyExpLabSys.common.sockets.**socket_server_status**()
Returns the status of all socket servers

Returns

Dict with port to status dict mapping. The status dict has the following keys: name, type, status (with str values) and since_last_activity with float value.

Return type dict

class PyExpLabSys.common.sockets.**PullUDPHandler**(*request*, *client_address*, *server*)
Bases: SocketServer.BaseRequestHandler

Request handler for the *DateDataPullSocket* and *DateDataPullSocket* socket servers. The commands this request handler understands are documented in the *handle()* method.

handle()
Returns data corresponding to the request

The handler understands the following commands:

COMMANDS

- **raw** (*str*): Returns all values on the form `x1,y1;x2,y2` in the order the codenames was given to the *DateDataPullSocket.__init__()* or *DataPullSocket.__init__()* method
- **json** (*str*): Return all values as a list of points (which in themselves are lists) e.g: `[[x1, y1], [x2, y2]]`), contained in a `json` string. The order is the same as in `raw`.
- **raw_wn** (*str*): (wn = with names) Same as raw but with names, e.g. `codenam1:x1,y1; codename2:x2,y2`. The order is the same as in `raw`.
- **json_wn** (*str*): (wn = with names) Return all data as a `dict` contained in a `json` string. In the dict the keys are the codenames.
- **codename#raw** (*str*): Return the value for `codename` on the form `x,y`
- **codename#json** (*str*): Return the value for `codename` as a list (e.g `[x1, y1]`) contained in a `json` string
- **codenames_raw** (*str*): Return the list of codenames on the form `name1,name2`
- **codenames_json** (*str*): Return a list of the codenames contained in a `json` string
- **name** (*str*): Return the name of the socket server
- **status** (*str*): Return the system status and status for all socket servers.

class PyExpLabSys.common.sockets.**CommonDataPullSocket**(*name*, *codenames*, *port*, *default_x*, *default_y*, *timeouts*, *check_activity*, *activity_timeout*, *init_timeouts=True*, *handler_class=<class PyExpLabSys.common.sockets.PullUDPHandler>*)

Bases: threading.Thread

Abstract class that implements common data pull socket functionality.

This common class is responsible for:

- Initializing the thread
- Checking the inputs
- Starting the socket server with the correct handler
- Initializing DATA with common attributes

__init__(*name*, *codenames*, *port*, *default_x*, *default_y*, *timeouts*, *check_activity*, *activity_timeout*, *init_timeouts=True*, *handler_class=<class PyExpLabSys.common.sockets.PullUDPHandler>*)

Initializes internal variables and data structure in the *DATA* module variable

Parameters

- **name** (*str*) – The name of the DataPullSocket server. Used for identification and therefore should contain enough information about location and purpose to unambiguously identify the socket server. E.g: `'DataPullSocket with data from giant laser on the moon'`
- **codenames** (*list*) – List of codenames for the measurements. The names must be unique and cannot contain the characters: `#,;:` and SPACE
- **port** (*int*) – Network port to use for the socket (deafult 9010)
- **default_x** (*float*) – The x value the measurements are initiated with
- **default_y** (*float*) – The y value the measurements are initiated with
- **timeouts** (*float or list of floats*) – The timeouts (in seconds as floats) the determines when the date data socket regards the data as being to old and reports that. If a list of timeouts is supplied there must be one value for each codename and in the same order.
- **init_timeouts** (*bool*) – Whether timeouts should be instantiated in the *DATA* module variable
- **handler_class** (*Sub-class of SocketServer.BaseRequestHandler*) – The UDP handler to use in the server
- **check_activity** (*bool*) – Whether the socket server should monitor activity. What detemines activity is described in the derived socket servers.
- **activity_timeout** (*float or int*) – The timespan in seconds which constitutes in-activity

run()

Starts the UPD socket server

stop()
Stops the UDP server

Note: Closing the server **and** deleting the socket server instance is necessary to free up the port for other usage

poke()
Pokes the socket server to let it know that there is activity

class PyExpLabSys.common.sockets.**DataPullSocket**(*name*, *codenames*, *port=9010*, *default_x=0.0*, *default_y=0.0*, *timeouts=None*, *check_activity=True*, *activity_timeout=900*, *poke_on_set=True*)
Bases: *PyExpLabSys.common.sockets.CommonDataPullSocket*

This class implements a UDP socket server for serving x, y type data. The UDP server uses the *PullUDPHandler* class to handle the UDP requests. The commands that can be used with this socket server are documented in the *PullUDPHandler.handle()* method.

__init__(*name*, *codenames*, *port=9010*, *default_x=0.0*, *default_y=0.0*, *timeouts=None*, *check_activity=True*, *activity_timeout=900*, *poke_on_set=True*)
Initializes internal variables and UPD server

For parameter description of `name`, `codenames`, `port`, `default_x`, `default_y`, `timeouts`, `check_activity` and `activity_timeout` see *CommonDataPullSocket.__init__()*.

Parameters **poke_on_set** (*bool*) – Whether to poke the socket server when a point is set, to let it know there is activity

set_point(*codename*, *point*, *timestamp=None*)
Sets the current point for codename

Parameters

- **codename** (*str*) – Name for the measurement whose current point should be set
- **value** (*iterable*) – Current point as a list or tuple of 2 floats: [x, y]
- **timestamp** (*float*) – A unix timestamp that indicates when the point was measured. If it is not set, it is assumed to be now. This value is used to evaluate if the point is new enough if timeouts are set.

class PyExpLabSys.common.sockets.**DateDataPullSocket**(*name*, *codenames*, *port=9000*, *default_x=0.0*, *default_y=0.0*, *timeouts=None*, *check_activity=True*, *activity_timeout=900*, *poke_on_set=True*)
Bases: *PyExpLabSys.common.sockets.CommonDataPullSocket*

This class implements a UDP socket server for serving data as a function of time. The UDP server uses the *PullUDPHandler* class to handle the UDP requests. The commands that can be used with this socket server are documented in the *PullUDPHandler.handle()* method.

__init__(*name*, *codenames*, *port=9000*, *default_x=0.0*, *default_y=0.0*, *timeouts=None*, *check_activity=True*, *activity_timeout=900*, *poke_on_set=True*)
Init internal variavles and UPD server

For parameter description of `name`, `codenames`, `port`, `default_x`, `default_y`, `timeouts`, `check_activity` and `activity_timeout` see *CommonDataPullSocket.__init__()*.

Parameters **poke_on_set** (*bool*) – Whether to poke the socket server when a point is set, to let it know there is activity

set_point_now(*codename*, *value*)
Sets the current y-value for codename using the current time as x

Parameters

- **codename** (*str*) – Name for the measurement whose current value should be set
- **value** (*float*) – y-value

set_point(*codename*, *point*)
Sets the current point for codename

Parameters

- **codename** (*str*) – Name for the measurement whose current point should be set
- **point** (*iterable*) – Current point as a list (or tuple) of 2 floats: [x, y]

class PyExpLabSys.common.sockets.**PushUDPHandler**(*request*, *client_address*, *server*)
Bases: SocketServer.BaseRequestHandler

This class handles the UDP requests for the *DataPushSocket*

handle()
Sets data corresponding to the request

The handler understands the following commands:

COMMANDS

- **json_wn#data** (*str*): Json with names. The data should be a JSON encoded dict with codename->value content. A complete command could look like:

  ```
  'json_wn#{"greeting":  "Live long and prosper", "number":  47}'
  ```

- **raw_wn#data** (*str*): Raw with names. The data should be a string with data on the following format: `codename1:type:data;codename2:type:data;...` where `type` is the type of the data and can be one of `'int'`, `'float'`, `'str'` and `'bool'`. NOTE; that neither the names or any data strings can contain any of the characters in *BAD_CHARS*. The `data` is a comma separated list of data items of that type. If there is more than one, they will be put in a list. An example of a complete raw_wn string could look like:

  ```
  'raw_wn#greeting:str:Live long and prosper;numbers:int:47,42'
  ```

- **name** (*str*): Return the name of the PushSocket server
- **status** (*str*): Return the system status and status for all socket servers.
- **commands** (*str*): Return a json encoded list of commands. The returns value is is prefixed with *PUSH_RET* and '#' so e.g. 'RET#actual_date'

class PyExpLabSys.common.sockets.**DataPushSocket**(*name*, *port=8500*, *action=u'store_last'*, *queue=None*, *callback=None*, *return_format=u'json'*, *check_activity=False*, *activity_timeout=900*)
Bases: threading.Thread

This class implements a data push socket and provides options for enqueuing, calling back or doing nothing on reciept of data

__init__(*name*, *port=8500*, *action=u'store_last'*, *queue=None*, *callback=None*, *return_format=u'json'*, *check_activity=False*, *activity_timeout=900*)
Initializes the DataPushSocket

Parameters

- **name** (*str*) – The name of the socket server. Used for identification and therefore should contain enough information about location and purpose to unambiguously identify the socket server. E.g: `'Driver push socket for giant laser on the moon'`
- **port** (*int*) – The network port to start the socket server on (default is 8500)
- **action** (*string*) – Determined the action performed on incoming data. The possible values are:
  - `'store_last'` (default and always) the incoming data will be stored, as a dict, only in the two properties; *last* and *updated*, where *last* contains only the data from the last reception and *updated* contains the newest value for each codename that has been received ever. Saving to these two properties **will always be done**, also with the other actions.
  - `'enqueue'`; the incoming data will also be enqueued
  - `'callback_async'` a callback function will also be called with the incoming data as an argument. The calls to the callback function will in this case happen asynchronously in a seperate thread
  - `'callback_direct'` a callback function will also be called and the result will be returned, provided it has a str representation. The return value format can be set with `return_format`
- **queue** (*Queue.Queue*) – If action is 'enqueue' and this value is set, it will be used as the data queue instead the default which is a new `Queue.Queue` instance without any further configuration.
- **callback** (*callable*) – A callable that will be called on incoming data. The callable should accept a single argument that is the data as a dictionary.
- **return_format** (*str*) – The return format used when sending callback return values back (used with the `'callback_direct'` action). The value can be:
  - `'json'`, which, if possible, will send the value back encoded as json
  - `'raw'` which, if possible, will encode a dict of values, a list of lists or None. If it is a dict, each value may be a list of values with same type, in the same way as they are received with the `'raw_wn'` command in the *PushUDPHandler.handle()* method. If the return value is a list of lists (useful e.g. for several data points), then **all** values must be of the same type. The format sent back looks like: `'1.0,42.0&1.5,45.6&2.0,47.0'`, where '&' separates the inner lists and ',' the points in those lists
  - `'string'` in which the string representation of the value the call back returns will be sent back. NOTE: These string representations may differ between Python 2 and 3, so do not parse them

run()
Starts the UPD socket server

stop()
Stops the UDP socket server

Note: Closing the server **and** deleting the `SocketServer.UDPServer` socket instance is necessary to free up the port for other usage

queue
Gets the queue, returns None if `action` is `'store_last'` or `'callback_direct'`

last
Gets a copy of the last data

Returns

tuple: `(last_data_time, last_data)` where `last_data` is the data from the last reception and `last_data_time` is the Unix timestamp of that reception. Returns `(None, None)` if no data has been recieved.

updated
Gets a copy of the updated total data, returns empty dict if no data has been received yet

Returns

`(updated_data_time, updated_data)` where `updated_data` is the total updated data after the last reception and `updated_data_time` is the Unix timestamp of that reception. Returns `(None, {})` if no data has been recieved.

Return type tuple

set_last_to_none()
Sets the last data point and last data point time to None

clear_updated()
Clears the total updated data and set the time of last update to None

poke()
Pokes the socket server to let it know that there is activity

class PyExpLabSys.common.sockets.**CallBackThread**(*queue*, *callback*)
Bases: `threading.Thread`

Class to handle the calling back for a DataReceiveSocket

__init__(*queue*, *callback*)
Initialize the local variables

Parameters

- **queue** (*Queue.Queue*) – The queue that queues up the arguments for the callback function
- **callback** (*callable*) – The callable that will be called when there are items in the queue

run()
Starts the calling back

stop()
Stops the calling back

exception PyExpLabSys.common.sockets.**PortStillReserved**
Bases: `exceptions.Exception`

Custom exception to explain socket server port still reserved even after closing the port

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

class PyExpLabSys.common.sockets.**LiveSocket**(*name*, *codenames*, *live_server=None*, *no_internal_data_pull_socket=False*, *internal_data_pull_socket_port=8000*)

Bases: object

This class implements a Live Socket

As of version 2 LiveSocket there are a few new features to note:

1. There is now support for values of any json-able object. The values can of course only be shown in a graph if they are numbers, but all other types can be shown in a table.
2. There is now support for generic xy data. Simply use *set_point()* or *set_batch()* and give it x, y values.

__init__(*name*, *codenames*, *live_server=None*, *no_internal_data_pull_socket=False*, *internal_data_pull_socket_port=8000*)
Intialize the LiveSocket

Parameters

- **name** (*str*) – The name of the socket
- **codenames** (*sequence*) – The codenames for the different data channels on this LiveSocket
- **live_server** (*sequence*) – 2 element sequence of hostname and port for the live server to connect to. Defaults to *(Settings.common_liveserver_host, Settings.common_liveserver_host)*.
- **no_internal_data_pull_socket** (*bool*) – Whether to not open an internal DataPullSocket. Defaults to False. See note below.
- **internal_data_pull_socket_port** (*int*) – Port for the internal DataPullSocket. Defaults to 8000. See note below.

Note: In general, any socket should also work as a status socket. But since the new design of the live socket, it no longers runs a UDP server, as would be required for it to work as a status socket. Therefore, LiveSocket now internally runs a DataPullSocket on port 8000 (that was the old LiveSocket port) to work as a status socket. With default setting, everything should work as before.

---

start()
Starts the internal DataPullSocket

stop()
Stop the internal DataPullSocket

set_batch(*data*)
Set a batch of points now

Parameters **data** (*dict*) – Batch of data on the form {codename1: (x1, y1), codename2: (x2, y2)}. Note, that for the live socket system, the y values need not be data in the form of a float, it can also be an int, bool or str. This is done to make it possible to also transmit e.g. equipment status to the live pages.

Note: All data is sent to the live socket proxy and onwards to the web browser clients as batches, so if the data is on batch form, might as well send it as such and reduce the number of transmissions.

set_batch_now(*data*)
Set a batch of point now

Parameters **data** (*dict*) – A mapping of codenames to values without times or x-values (see example below)

The format for data is:

```
{'measurement1': 47.0, 'measurement2': 42.0}
```

set_point_now(*codename*, *value*)
Sets the current value for codename using the current time as x

Parameters

- **codename** (*str*) – Name for the measurement whose current value should be set
- **value** (*float, int, bool or str*) – value

set_point(*codename*, *point*)
Sets the current point for codename

Parameters

- **codename** (*str*) – Name for the measurement whose current point should be set
- **point** (*list or tuple*) – Current value "point" as a list (or tuple) of items, the first must be a float, the second can be float, int, bool or str

reset(*codenames*)
Send the reset signal for codenames

Parameters **codenames** (*list*) – List of codenames

PyExpLabSys.common.sockets.**BAD_CHARS = [u'#', u',', u';', u':', u'&']**
The list of characters that are not allowed in code names

PyExpLabSys.common.sockets.**UNKNOWN_COMMAND = u'UNKNOWN_COMMMAND'**
The string returned if an unknown command is sent to the socket

PyExpLabSys.common.sockets.**OLD_DATA = u'OLD_DATA'**
The string used to indicate old or obsoleted data

PyExpLabSys.common.sockets.**PUSH_ERROR = u'ERROR'**
The answer prefix used when a push failed

PyExpLabSys.common.sockets.**PUSH_ACK = u'ACK'**
The answer prefix used when a push succeds

PyExpLabSys.common.sockets.**PUSH_EXCEP = u'EXCEP'**
The answer prefix for when a callback or callback value formatting produces an exception

PyExpLabSys.common.sockets.**PUSH_RET = u'RET'**
The answer prefix for a callback return value

PyExpLabSys.common.sockets.**DATA = {}**
The variable used to contain all the data.

The format of the DATA variable is the following. The DATA variable is a dict, where each key is an integer port number and the value is the data for the socket server on that port. The data for each individual socket server is always a dict, but the contained values will depend on which kind of socket server it is, Examples below.

For a *DateDataPullSocket* the dict will resemble this example:

```
{'activity': {'activity_timeout': 900,
              'check_activity': True,
              'last_activity': 1413983209.82526},
 'codenames': ['var1'],
 'data': {'var1': (0.0, 0.0)},
 'name': 'my_socket',
 'timeouts': {'var1': None},
 'type': 'date'}
```

For a *DataPullSocket* the dict will resemble this example:

```
{'activity': {'activity_timeout': 900,
              'check_activity': True,
              'last_activity': 1413983209.825451},
 'codenames': ['var1'],
 'data': {'var1': (0.0, 0.0)},
 'name': 'my_data_socket',
 'timeouts': {'var1': None},
 'timestamps': {'var1': 0.0},
 'type': 'data'}
```

For a *DataPushSocket* the dict will resemble this example:

```
{'action': 'store_last',
 'activity': {'activity_timeout': 900,
              'check_activity': False,
              'last_activity': 1413983209.825681},
 'last': None,
 'last_time': None,
 'name': 'my_push_socket',
 'type': 'push',
 'updated': {},
 'updated_time': None}
```

PyExpLabSys.common.sockets.**TYPE_FROM_STRING = {u'bool': <function bool_translate at 0x7fdb**

The dict that transforms strings to convertion functions

PyExpLabSys.common.sockets.**run_module**()

This functions sets

3.5 The socket_client module

3.6 The utilities module

This module contains convenience functions for quick setup of common tasks.

Table of Contents

3.6.1 Get a logger

The *get_logger()* function is a convenience function to setup logging output. It will return a named logger, which can be used inside programs. The function has the ability to setup logging both to a terminal, to a log file, including setting up log rotation and for sending out email on log message at warning level or above.

3.6.2 get_logger usage examples

Logging to terminal and send emails on warnings and above (default)

To get a named logger that will output logging information to the terminal and send emails on warnings and above, do the following:

```
from PyExpLabSys.common.utilities import get_logger
LOGGER = get_logger('name_of_my_logger')
```

where the `name_of_my_logger` should be some descriptive name for what the program/script does e.g. "coffee_machine_count_monitor".

From the returned `LOGGER`, information can now be logged via the usual `.info()`, `.warning()` methods etc.

To turn logging to the terminal of (and only use the other configured logging handlers), set the optional boolean `terminal_log` parameter to `False`.

The email notification on warnings and above is on-by-default and is controlled by the two optional boolean parameters `email_on_warnings` and `email_on_errors`. It will send emails on logged warnings to the warnings list and on logged errors (and above) to the error list.

Rotating file logger

To get a named logger that also logs to a file do:

```
from PyExpLabSys.common.utilities import get_logger
LOGGER = get_logger('name_of_my_logger', file_log=True)
```

The log will be written to the file `name_of_my_logger.log`. The file name can be changed via the option `file_name`, the same way as the maximum log file size and number of saved backups can be changed, as documented *below*.

Getting the logger to send emails on un-caught exceptions

To also make the logger send en email, containing any un-caught exception, do the following:

```
from PyExpLabSys.common.utilities import get_logger
LOGGER = get_logger('Test logger')

def main():
    pass  # All your main code that might raise exceptions

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass  # Shut down code here
    except Exception:
        LOGGER.exception("Main program failed")
        # Possibly shut down code here
        raise
```

Note: The argument to `LOGGER.exception` is a string, just like all other `LOGGER` methods. All the exception information, like the traceback, line number etc., is picked up automatically by the logger. Also note, that the `raise` will re-raise the caught exception, to make sure that the program fails like it is supposed to. That it is the caught exception that is re-raised, is implicit when using raise without arguments in an except clause.

Note: In the example there is a seperate `except` for the `KeyboardInterrupt` exception, to make it possible to use keyboard interrupts to shut the program down, without sending an exception email about it.

3.6.3 Auto-generated module documentation

This module contains a convenience function for easily setting up a logger with the `logging` module.

This module uses the following settings from the *`Settings`* class:

- util_log_warning_email
- util_log_error_email
- util_log_mail_host
- util_log_max_emails_per_period (defaults to 5)
- util_log_email_throttle_time (defaults to 86400s = 1day)
- util_log_backlog_limit (defaults to 250)

Note: All of these settings are at present read from the settings module at import time, so if it is desired to modify them at run time, it should be done before import

PyExpLabSys.common.utilities.**SETTINGS = <PyExpLabSys.settings.Settings object>**
The *`Settings`* object used in this module

PyExpLabSys.common.utilities.**WARNING_EMAIL = 'FYS-list-CINF-FM@fysik.dtu.dk'**
The email list warning emails are sent to

PyExpLabSys.common.utilities.**ERROR_EMAIL = 'FYS-list-CINF-FM@fysik.dtu.dk'**
The email list error emails are sent to

PyExpLabSys.common.utilities.**MAIL_HOST = 'mail.fysik.dtu.dk'**
The email host used to send emails on logged warnings and errors

PyExpLabSys.common.utilities.**MAX_EMAILS_PER_PERIOD = 5**
The maximum number of emails the logger will send in *EMAIL_THROTTLE_TIME*

PyExpLabSys.common.utilities.**EMAIL_THROTTLE_TIME = 86400**
The time period that the numbers of emails will be limited within

PyExpLabSys.common.utilities.**EMAIL_BACKLOG_LIMIT = 250**
The maximum number of messages in the email backlog that will be sent when the next email is let through

PyExpLabSys.common.utilities.**get_logger**(*name*, *level=u'INFO'*, *terminal_log=True*, *file_log=False*, *file_name=None*, *file_max_bytes=1048576*, *file_backup_count=1*, *email_on_warnings=True*, *email_on_errors=True*)

Setup and return a program logger

This is meant as a logger to be used in a top level program/script. The logger is set up for with terminal, file and email handlers if requested.

Parameters

- **name** (*str*) – The name of the logger, e.g: 'my_fancy_program'. Passing in an empty string will return the root logger. See note below.
- **level** (*str*) – The level for the logger. Can be either `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'` or `'CRITICAL'`. See `logging` for details. Default is `'INFO'`.
- **terminal_log** (*bool*) – If `True` then logging to a terminal will be activated. Default is `True`.
- **file_log** (*bool*) – If `True` then logging to a file, with log rotation, will be activated. If `file_name` is not given, then `name + '.log'` will be used. Default is `False`.
- **file_name** (*str*) – Optional file name to log to
- **file_max_size** (*int*) – The maximum size of the log file in bytes. The default is `1048576` (1MB), which corresponds to roughly 10000 lines of log per file.
- **file_backup_count** (*int*) – The number of backup logs to keep. The default is `1`.
- **email_on_warnings** (*bool*) – Whether to send an email to the *WARNING_EMAIL* email list if a warning is logged. The default is `True`.
- **email_on_error** (*bool*) – Whether to send en email to the *ERROR_EMAIL* email list if an error (or any logging level above) is logged. The default is `True`.

Returns A logger module with the requested setup

Return type `logging.Logger`

Note: Passing in the empty string as the `name`, will return the root logger. That means that all other library loggers will inherit the level and handlers from this logger, which may potentially result in **a lot** of output. See *activate_library_logging()* for a way to activate the library loggers from PyExpLabSys in a more controlled manner.

PyExpLabSys.common.utilities.**get_library_logger_names**()
Return all loggers currently configured in PyExpLabSys

PyExpLabSys.common.utilities.**print_library_logger_names**()
Nicely printout all loggers currently configured in PyExpLabSys

PyExpLabSys.common.utilities.**activate_library_logging**(*logger_name*, *logger_to_inherit_from=None*, *level=None*, *terminal_log=True*, *file_log=False*, *file_name=None*, *file_max_bytes=1048576*, *file_backup_count=1*, *email_on_warnings=True*, *email_on_errors=True*)

Activate logging for a PyExpLabSys library logger

Parameters

- **logger_name** (*str*) – The name of the logger to activate, as returned by *get_library_logger_names()*
- **logger_to_inherit_from** (*logging.Logger*) – (Optional) If this is set, the library logger will simply share the handlers that are present in this logger. The library to be activated will also inherit the level from this logger, unless `level` is set, in which case it will override. In case neither `level` nor the level on `logger_to_inherit_from` is set, the level will not be changed.
- **level** (*str*) – (Optional) See docstring for *get_logger()*. If `logger_to_inherit_from` is not set, it will default to 'info'.
- **terminal_log** (*bool*) – See docstring for *get_logger()*
- **file_log** (*bool*) – See docstring for *get_logger()*
- **file_name** (*str*) – See docstring for *get_logger()*
- **file_max_size** (*int*) – See docstring for *get_logger()*
- **file_backup_count** (*int*) – See docstring for *get_logger()*
- **email_on_warnings** (*bool*) – See docstring for *get_logger()*
- **email_on_error** (*bool*) – See docstring for *get_logger()*

class PyExpLabSys.common.utilities.**CustomSMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*)

Bases: `logging.handlers.SMTPHandler`

PyExpLabSys modified SMTP handler

emit(*record*)
Custom emit that throttles the number of email sent

getSubject(*record*)
Returns subject with hostname

class PyExpLabSys.common.utilities.**CustomSMTPWarningHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*)

Bases: *PyExpLabSys.common.utilities.CustomSMTPHandler*

Custom SMTP handler to emit record only if: warning =< level < error

emit(*record*)
Custom emit that checks if: warning =< level < error

PyExpLabSys.common.utilities.**call_spec_string**()
Return the argument names and values of the method or function it was called from as a string

Returns

The argument string, e.g: (name='hello', codenames=['aa', 'bb'], port=8000)

Return type str

3.7 The text plot module

3.7.1 Autogenerated API documentation for text_plot

GNU plot based plots direcly into a curses window

This module provides two classes for creating text plots, one for creating a text plot as a text string *AsciiPlot* and one for writing the resulting plot directly into a curses window, *CursesAsciiPlot*, automatically adjusting the size of the plot to the window.

AsciiPlot example:

```
from PyExpLabSys.common.text_plot import AsciiPlot
import numpy

x = numpy.linspace(0, 6.28, 100)
y = numpy.sin(x)

ascii_plot = AsciiPlot(title='Sine', xlabel='x', ylabel='y', size=(80, 24))
text_plot = ascii_plot.plot(x, y)
print(text_plot)
```

Produces the following output:

```
                                    Sine

    1 +-------------------------------------------------------------------------+
      |          ***       ***         +         +          +          +       |
  0.8 |-+      **              **                                             +-|
  0.6 |-+    ***                 *                                             +-|
      |     **                    **                                            |
  0.4 |-+ *                       **                                            +-|
      |   *                         *                                           |
  0.2 |**                           *                                           +-|
    0 |*+                           **                                          +-|
y     |                               **                                  **      |
 -0.2 |-+                               *                                **     +-|
      |                                  *                              *       |
 -0.4 |-+                                 **                           *       +-|
      |                                    **                      **           |
 -0.6 |-+                                    *                   ***           +-|
 -0.8 |-+                                     **               **               +-|
```

(continues on next page)

(continued from previous page)

```
      |          +          +          +         + ***      +***       +          |
   -1 +-------------------------------------------------------------------------+
      0          1          2          3         4          5          6          7
                                        x
```

`CursesAsciiPlot` example

```
import time
import curses
import numpy as np
from PyExpLabSys.common.text_plot import CursesAsciiPlot

# Init and clear
stdscr = curses.initscr()
stdscr.clear()
curses.noecho()

# Make plot window
screen_size = stdscr.getmaxyx()
# Make the plot a little smaller than the main window, to allow a
# littel space for text at the top
win = curses.newwin(screen_size[0] - 3, screen_size[1], 3, 0)

t_start = time.time()
try:
    # Create the Curses Ascii Plotter
    ap = CursesAsciiPlot(
        win, title="Log of sine of time + 1.1", xlabel="Time [s]",
        logscale=True,
    )

    # Plot the sine to time since start and 10 sec a head
    while True:
        stdscr.clear()
        t0 = time.time() - t_start
        # Write the time right now to the main window
        stdscr.addstr(1, 3, 'T0: {:.2f}       '.format(t0))
        stdscr.refresh()
        x = np.linspace(t0, t0 + 10)
        y = np.sin(x) + 1.1
        ap.plot(x, y, legend="Sine")
        time.sleep(0.2)
finally:
    curses.echo()
    curses.endwin()
```

class PyExpLabSys.common.text_plot.**CursesAsciiPlot**(*curses_win*, ***kwargs*)

Bases: object

A Curses Ascii Plot

__init__(*curses_win*, ***kwargs*)

Initialize local variables

Parameters **curses_win** (*curses-window-objects*) – The curses window to print the plot into

For possible value for kwargs, see arguments for *AsciiPlot.__init__()*.

plot (**args*, ***kwargs*)
Plot data to the curses window

For an explanation of the arguments, see *AsciiPlot.plot()*.

class PyExpLabSys.common.text_plot.**AsciiPlot**(*title=None*, *xlabel=None*, *ylabel=None*, *logscale=False*, *size=(80*, *24)*, *debug=False*)

Bases: object

An Ascii Plot

__init__ (*title=None*, *xlabel=None*, *ylabel=None*, *logscale=False*, *size=(80*, *24)*, *debug=False*)
Initialize local varibles

Parameters

- **title** (*str*) – The title of the plot if required
- **xlabel** (*str*) – The xlabel of the plot if required
- **ylabel** (*str*) – The ylabel of the plot if required
- **logscale** (*bool*) – If the yaxis should use log scale
- **size** (*tuple*) – A list or tuple with two integers indication the x and y size (i.e. number of columns and lines) of the plot
- **debug** (*bool*) – Whether to show the command sent to gnuplot

write (*string*)
Write string to gnuplot

String must be n terminated

plot (*x*, *y*, *style='lines'*, *legend=''*)
Plot data

Parameters

- **x** (*iterable*) – An iterable of floats or ints to plot
- **y** (*iterable*) – An iterable of floats or ints to plot
- **style** (*str*) – 'lines' or 'points'
- **legend** (*str*) – The legend of the data (leave to empty string to skip)

3.8 The combos module

The combos are cobinations of other PyExpLabSys components in commonly used configurations. Currently the only implemented combination are:

- *LiveContinuousLogger* which combines a *LiveSocket* with a ContinuousDataSaver and

Table of Contents

3.8.1 Examples

LiveContinuousLogger

This example shows the case of using the combo to log values individually, using the "now" variety of the method (*LiveContinuousLogger.log_point_now()*), which uses the time now as the x-value:

```
from time import sleep
from random import random
from math import sin

from PyExpLabSys.combos import LiveContinuousLogger

# Initialize the combo and start it
combo = LiveContinuousLogger(
    name='test',
    codenames=['dummy_sine_one', 'dummy_sine_two'],
    continuous_data_table='dateplots_dummy',
    username='dummy',
    password='dummy',
    time_criteria=0.1,
)
combo.start()

# Measurement loop (typically runs forever, here just for 10 sec)
for _ in range(10):
    # The two sine values here emulate a value to be logged
    sine_one = sin(random())
    sine_two = sin(random())
    combo.log_point_now('dummy_sine_one', sine_one)
    combo.log_point_now('dummy_sine_two', sine_two)
    sleep(1)

combo.stop()
```

or if it is preferred to keep track of the timestamp manually, the *LiveContinuousLogger.log_point()* method which can be used instead:

```
from time import sleep, time
from math import sin, pi

from PyExpLabSys.combos import LiveContinuousLogger

# Initialize the combo and start it
combo = LiveContinuousLogger(
    name='test',
    codenames=['dummy_sine_one', 'dummy_sine_two'],
    continuous_data_table='dateplots_dummy',
    username='dummy',
    password='dummy',
    time_criteria=0.1,
```

(continues on next page)

(continued from previous page)

```
)
combo.start()

# Measurement loop (typically runs forever, here just for 10 sec)
for _ in range(10):
    # The two sine values here emulate a value to be logged
    now = time()
    sine_one = sin(now)
    sine_two = sin(now + pi)
    combo.log_point('dummy_sine_one', (now, sine_one))
    combo.log_point('dummy_sine_two', (now, sine_two))
    sleep(1)

combo.stop()
```

Of course, like most of the underlying code, the combo also takes data at batches, as shown in the following example (using *LiveContinuousLogger.log_batch()*):

```
from time import sleep, time
from math import sin, pi

from PyExpLabSys.combos import LiveContinuousLogger

# Initialize the combo and start it
combo = LiveContinuousLogger(
    name='test',
    codenames=['dummy_sine_one', 'dummy_sine_two'],
    continuous_data_table='dateplots_dummy',
    username='dummy',
    password='dummy',
    time_criteria=0.1,
)
combo.start()

# Measurement loop (typically runs forever, here just for 10 sec)
for _ in range(10):
    # The two sine values here emulate a value to be logged
    now = time()
    points = {
        'dummy_sine_one': (now, sin(now)),
        'dummy_sine_two': (now, sin(now + pi)),
    }
    combo.log_batch(points)
    sleep(1)

combo.stop()
```

For the bacthes there is of course also a "now" variety (*LiveContinuousLogger.log_batch_now()*), which there is no example for, but the difference is same as for the single point/value.

3.8.2 Auto-generated module documentation

This module contains socket, database saver and logger heuristic combinations

class PyExpLabSys.combos.**LiveContinuousLogger**(*name*, *codenames*, *continuous_data_table*, *username*, *password*, *time_criteria=None*, *absolute_criteria=None*, *relative_criteria=None*, *live_server_kwargs=None*)

Bases: object

A combination of a *LiveSocket* and a *ContinuousDataSaver* that also does logging heuristics

FIXME explain the term log

__init__(*name*, *codenames*, *continuous_data_table*, *username*, *password*, *time_criteria=None*, *absolute_criteria=None*, *relative_criteria=None*, *live_server_kwargs=None*)

Initialize local data

Parameters

- **name** (*str*) – The name to be used in the sockets
- **codenames** (*sequence*) – A sequence of codenames. These codenames are the measurements codenames for the *ContinuousDataSaver* and they will also be used as the codenames for the *LiveSocket*.
- **continuous_data_table** (*str*) – The continuous data table to log data to
- **username** (*str*) – The MySQL username
- **password** (*str*) – The password for `username` in the database
- **time_criteria** (*float or dict*) – (Optional) The time after which a point will always be saved in the database. Either a single value, which will be used for all codenames or a dict of codenames to values. If supplying a dict, each codename must be present as a key.
- **absolute_criteria** (*float or dict*) – (Optional) The absolute value difference criteria. Either a single value or one for each codename, see *time_criteria* for details.
- **relative_criteria** (*float or dict*) – (Optional) The relative value difference criteria. Either a single value or one for each codename, see *time_criteria* for details.
- **lives_server_kwargs** (*dict*) – (Optinal) A dict of keyword arguments for the *LiveSocket*. See the doc string for *LiveSocket.__init__()* for additional details.

start()

Start the underlying *LiveSocket* and *ContinuousDataSaver*

stop()

Stop the underlying *LiveSocket* and *ContinuousDataSaver*

log_point_now(*codename*, *value*)

Log a point now

As the time will be attached the time now.

For an explanation of what is meant by the term "log", see the *class docstring*.

Parameters

- **codename** (*str*) – The codename to log this point for
- **value** (*float*) – The value to store with the time now (`time.time()`)

log_point (*codename*, *point*)
Log a point

For an explanation of what is meant by the term "log", see the *class docstring*.

Parameters

- **codename** (*str*) – The codename to log this point for
- **point** (*sequence*) – A (unix_time, value) two item sequence (e.g. list or tuple), that represents a point

log_batch_now (*values*)
Log a batch of values now

For an explanation of what is meant by the term "log", see the *class docstring*.

Parameters **values** (*dict*) – Dict of codenames to values. The values will be stored with the time now (`time.time()`)

log_batch (*points*)
Log a batch of points

For an explanation of what is meant by the term "log", see the *class docstring*.

Parameters **points** (*dict*) – Dict of codenames to points

3.9 The settings module

Settings for PyExpLabSys component are handled via the *Settings* class in the `settings` module.

3.9.1 Getting started

To use the settings module instantiate a *Settings* object and access the settings as attributes:

```
>>> from PyExpLabSys.settings import Settings
>>> settings = Settings()
>>> settings.util_log_max_emails_per_period
5
```

User settings can be modified at run time simply by assigning a new value to the attributes:

```
>>> settings.util_log_max_emails_per_period = 7
>>> settings.util_log_max_emails_per_period
7
```

3.9.2 Details

The settings are handled in two layers; **defaults** and **user settings**.

The defaults are stored in the PyExpLabSys/defaults.yaml file.

Note: It is not possible to write to a setting that does not have a default

Note: In the defaults, a value of `null` is used to indicate a settings that must be overwritten by a user setting before any modules tries to use it.

The user settings are stored in a user editable file. The path used is stored in the *settings.USERSETTINGS_PATH* variable. On Linux system the user settings path is `~/.config.PyExpLabSys.user_settings.yaml`.

Note: If the value is None, it means that your operating system is not yet supported by the sett ings module. This should be reported as an issue on Github.

All *Settings* objects share the same settings, so changes made via one object will be used everywhere, in fact that is what makes it possible to modify settings at runtime (as shown above). **Do however note, that different modules reads the settings at different points in time. Some will read them when an object from that module is instantiated and others will read them at module import time**. That means, that for some modules it will be necessary to modify the settings before the rest of the PyExpLabSys modules is imported, in order to be able to modify them at runtime. At which point in time the settings are read should be stated in the module documentation.

3.9.3 Auto-generated module documentation

This module contains the modules used for settings for PyExpLabSys

To use the settings module instantiate a *Settings* object and access the settings as attributes:

```
>>> from PyExpLabSys.settings import Settings
>>> settings = Settings()
>>> settings.util_log_max_emails_per_period
5
```

The settings in the *Settings* are formed by 2 layers. The bottom layer are the defaults, that are stored in the PyExpLabSys/defaults.yaml file. Op top of those are placed the user settings, that originate from the file whose path is in the *settings.USERSETTINGS_PATH* variable. The user settings can me modified at run time as opposed to having to write them to the user settings file before running. This is done simply by writing to the properties on the settings object:

```
>>> settings.util_log_max_emails_per_period = 7
>>> settings.util_log_max_emails_per_period
7
```

All *Settings* objects share the same settings, so these changes will be used when using other parts of PyExpLabSys that makes use of one of the settings. Do however note, that different parts of PyExpLabSys use the settings at different times (instantiate, call etc.) so check with the documentation for each component when the settings needs to be modified to take effect.

PyExpLabSys.settings.**value_str**(*obj*)
Return a object and type str or NOT_SET if obj is None

class PyExpLabSys.settings.**Settings**
Bases: object

The PyExpLabSys settings object

The settings are available to get and setable on this object as attributes i.e:

```
>>> from PyExpLabSys.settings import Settings
>>> settings = Settings()
>>> settings.util_log_max_emails_per_period
5
```

The settings are stored as a ChainMap of the defaults and the user settings and this ChainMap object containing the current state of the settings is shared between all *Settings* objects.

To get a list of all available settings see the *Settings.settings_names* attribute. To get a pretty print of all settings names, types, default values, user setting values (if any) use the *Settings.print_settings()* method.

settings = ChainMap({'util_log_warning_email': 'FYS-list-CINF-FM@fysik.dtu.dk', 'util_
The settings ChainMap

settings_names = ['common_sql_reader_password', 'util_log_warning_email', 'util_log_er
The available setting names

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

print_settings()
Pretty print of all default and user settings

PyExpLabSys.settings.**main**()
Main function used to simple testing

CHAPTER 4

File parsers

File parsers.

Table of Contents

4.1 XML based file formats

4.1.1 Specs File Format

This file is used to parse XPS and ISS data from XML files from the SPECS program.

In this file format the spectra (called regions) are containd in region groups inside the files. This structure is mirrored in the data structure below where classes are provided for the 3 top level objects:

Files -> Region Groups -> Regions

The parser is strict, in the sense that it will throw an exception if it encounters anything it does not understand. To change this behavior set the EXCEPTION_ON_UNHANDLED module variable to False.

Usage examples

To use the file parse, simply feed the top level data structure a path to a data file and start to use it:

```
from PyExpLabSys.file_parsers.specs import SpecsFile
import matplotlib.pyplot as plt

file_ = SpecsFile('path_to_my_xps_file.xml')
# Access the regions groups by iteration
for region_group in file_:
    print '{} regions groups in region group: {}'.format(
        len(region_group), region_group.name)

# or by index
region_group = file_[0]

# And again access regions by iteration
for region in region_group:
    print 'region: {}'.format(region.name)

# or by index
region = region_group[0]

# or you can search for them from the file level
region = list(file_.search_regions('Mo'))[0]
print region
# NOTE the search_regions method returns a generator of results, hence the
# conversion to list and subsequent indexing

# From the regions, the x data can be accessed either as kinetic
# or binding energy (for XPS only) and the y data can be accessed
# as averages of the counts, either as pure count numbers or as
# counts per second. These options works independently of each
# other.

# counts as function of kinetic energy
plt.plot(region.x, region.y_avg_counts)
plt.show()

# cps as function of binding energy
plt.plot(region.x_be, region.y_avg_cps)
plt.show()

# Files also have a useful str representation that shows the hierachi
print file_
```

Notes

The file format seems to basically be a dump, of a large low level data structure from the implementation language. With an appropriate mapping of low level data structure types to python types (see details below and in the simple_convert function), this data structure could have been mapped in its entirety to python types, but in order to provide a more clear data structure a more object oriented approach has been taken, where the top most level data structures are implemented as classes. Inside of these classes, the data is parsed into numpy arrays and the remaining low level data structures are parsed in python data structures with the simple_convert function.

Module Documentation

PyExpLabSys.file_parsers.specs.**simple_convert**(*element*)
Converts a XML data structure to pure python types.

Parameters **element** (*xml.etree.ElementTree.Element*) – The XML element to convert

Returns A hierachi of python data structure

Return type object

Simple element types are converted as follows:

XML type	Python type
string	str
ulong	long
double	float
boolean	bool
struct	dict
sequence	list

Arrays are converted to numpy arrays, wherein the type conversion is:

XML type	Python type
ulong	numpy.uint64
double	numpy.double

Besides these types there are a few special elements that have a custom conversion.

- **Enum** are simply converted into their value, since enums are considered to be a program implementation detail whose information is not relavant for a data file parser
- **Any** is skipped and replaced with its content

class PyExpLabSys.file_parsers.specs.**SpecsFile**(*filepath*, *encoding=None*)
Bases: list

This is the top structure for a parsed file which represents a list of RegionGroups

The class contains a 'filepath' attribute.

__init__(*filepath*, *encoding=None*)
Parse the XML and initialize the internal variables

regions_iter
Returns a iteration over the regions

search_regions_iter(*search_term*)
Returns an generator of search results for regions by name

Parameters **search_term** (*str*) – The term to search for (case sensitively)

Returns An iterator of maching regions

Return type generator

search_regions(*search_term*)
Returns an list of search results for regions by name

Parameters **search_term** (*str*) – The term to search for (case sensitively)

Returns A list of matching regions

Return type list

unix_timestamp
Returns the unix timestamp of the first region

get_analysis_method()
Returns the analysis method of the file

Raises **ValueError** – If more than one analysis method is used

class PyExpLabSys.file_parsers.specs.**RegionGroup**(*xml*)
Bases: list

Class that represents a region group, which consist of a list of regions

The class contains a 'name' and and 'parameters' attribute.

__init__(*xml*)
Initializes the region group

Expects to find 3 subelement; the name, regions and parameters. Anything else raises an exception.

Parsing parameters is not supported and therefore logs a warning if there are any.

class PyExpLabSys.file_parsers.specs.**Region**(*xml*)
Bases: object

Class that represents a region

The class contains attributes for the items listed in the 'information_names' class variable.

Some useful ones are:

- **name**: The name of the region
- **region**: Contains information like, dwell_time, analysis_method, scan_delta, excitation_energy etc.

All auxiliary information is also available from the 'info' attribute.

__init__(*xml*)
Parse the XML and initialize internal variables

Parameters **xml** (*xml.etree.ElementTree.Element*) – The region XML element

x
Returns the kinetic energy x-values as a Numpy array

x_be
Returns the binding energy x-values as a Numpy array

iter_cycles
Returns a generator of cycles

Each cycle is in itself a generator of lists of scans. To iterate over single scans do:

```
for cycle in self.iter_cycles:
    for scans in cycle:
        for scan in scans:
            print scan
```

or use *iter_scans*, which do just that.

iter_scans
Returns an generator of single scans, which in themselves are Numpy arrays

y_avg_counts
Returns the average counts as a Numpy array

y_avg_cps
Returns the average counts per second as a Numpy array

unix_timestamp
Returns the unix timestamp of the first cycle

exception PyExpLabSys.file_parsers.specs.**NotXPSException**
Bases: exceptions.Exception

Exception for trying to interpret non-XPS data as XPS data

4.2 Binary File Formats

File parser for Chemstation files

Note: This file parser went through a large re-write on ??? which changed the data structures of the resulting objects. This means that upon upgrading it *will* be necessary to update code. The re-write was done to fix some serious errors from the first version, like relying on the Report.TXT file for injections summaries. These are now fetched from the more ordered CSV files.

---

exception PyExpLabSys.file_parsers.chemstation.**NoInjections**
Bases: exceptions.Exception

Exception raised when there are no injections in the sequence

class PyExpLabSys.file_parsers.chemstation.**Sequence**(*sequence_dir_path*)
Bases: object

The Sequence class for the Chemstation data format

Parameters

- **injections** (*list*) – List of *Injection*'s in this sequence
- **sequence_dir_path** (*str*) – The path of this sequence directory
- **metadata** (*dict*) – Dict of metadata

__init__(*sequence_dir_path*)
Instantiate object properties

Parameters **sequence_dir_path** (*str*) – The path of the sequence

full_sequence_dataset(*column_names=None*)
Generate peak name specific dataset

This will collect area values for named peaks as a function of time over the different injections.

Parameters **column_names** (*dict*) – A dict of the column names needed from the report lines. The dict should hold the keys: 'peak_name', 'retention_time' and 'area'. It defaults to: *column_names = {'peak_name': 'Compound Name', 'retention_time': 'Retention Timemin', 'area': 'Area'}*

Returns Mapping of signal_and_peak names and the values

Return type dict

class PyExpLabSys.file_parsers.chemstation.**Injection**(*injection_dirpath*, *load_raw_spectra=True*, *read_report_txt=True*)

Bases: object

The Injection class for the Chemstation data format

Parameters

- **injection_dirpath** (*str*) – The path of the directory of this injection
- **reports** (*defaultdict*) – Signal -> list_of_report_lines dict. Each report line is dict of column headers to type converted column content. E.g:

```
{u'Area': 22.81, u'Area %': 0.24, u'Height': 12.66,
 u'Peak Number': 1, u'Peak Type': u'BB', u'Peak Widthmin':
 0.027, u'Retention Timemin': 5.81}
```

  The columns headers are also stored in :attr'~metadata' under the *columns* key.

- **reports_raw** (*defaultdict*) – Same as `reports` except the content is not type converted.
- **metadata** (*dict*) – Dict of metadata
- **raw_files** (*dict*) – Mapping of ch_file_name -> *CHFile* objects
- **report_txt** (*str or None*) – The content of the Report.TXT file from the injection folder is any

__init__(*injection_dirpath*, *load_raw_spectra=True*, *read_report_txt=True*)

Instantiate Injection object

Parameters

- **injection_dirpath** (*str*) – The path of the injection directory
- **load_raw_spectra** (*bool*) – Whether to load raw spectra or not
- **read_report_txt** (*bool*) – Whether to read and save the Report.TXT file

PyExpLabSys.file_parsers.chemstation.**parse_utf16_string**(*file_*, *encoding=u'UTF16'*)

Parse a pascal type UTF16 encoded string from a binary file object

class PyExpLabSys.file_parsers.chemstation.**CHFile**(*filepath*)

Bases: object

Class that implementats the Agilent .ch file format version 179

Warning: Not all aspects of the file header is understood, so there may and probably is information that is not parsed. See the method `_parse_header_status()` for an overview of which parts of the header is understood.

Note: Although the fundamental storage of the actual data has change, lots of inspiration for the parsing of the header has been drawn from the parser in the ImportAgilent.m file in the chemplexity/chromatography project project. All credit for the parts of the header parsing that could be reused goes to the author of that project.

values

The internsity values (y-value) or the spectrum. The unit for the values is given in *metadata['units']*

Type numpy.array

metadata
The extracted metadata

Type dict

filepath
The filepath this object was loaded from

Type str

__init__ (*filepath*)
Instantiate object

Parameters **filepath** (*str*) – The path of the data file

times
The time values (x-value) for the data set in minutes

CHAPTER 5

Apps

This section documents the various apps in PyExpLabSys.

5.1 The Bakeout App

Contents

This app ... FIXME

5.1.1 A Section

A subsection

CHAPTER 6

Hardware Drivers

This section documents the hardware drivers developed at CINF. Most of the drivers are for equipment to surface science such as mass spectrometers and pressure gauges, but there are also some drivers for more general equipment like temperature read out units.

6.1 The bio_logic module

This module implements a driver for the SP-150 BioLogic potentiostat.

The implementation is built up around the notion of an instrument and techniques. To communicate with the device, it is required to first create an instrument, and then a technique and then load the technique onto the instrument. This implementation was chosen because it closely reflects that way the *specification* is written and the official ECLab program is structured.

This driver communicates with the potentiostats via the EC-lib dll, which is present in the ECLab development packages. This package must be installed before the driver can be used. It can be downloaded from the BioLogic website.

See the *Usage Example* sections some examples on how to use this driver.

See the *Inheritance diagram* for an inheritance diagram, that gives a good overview over the available instruments and techniques.

> **Note:** See some important notes on 64 bit Windows and instruments series in the beginning of the *API documentation*.

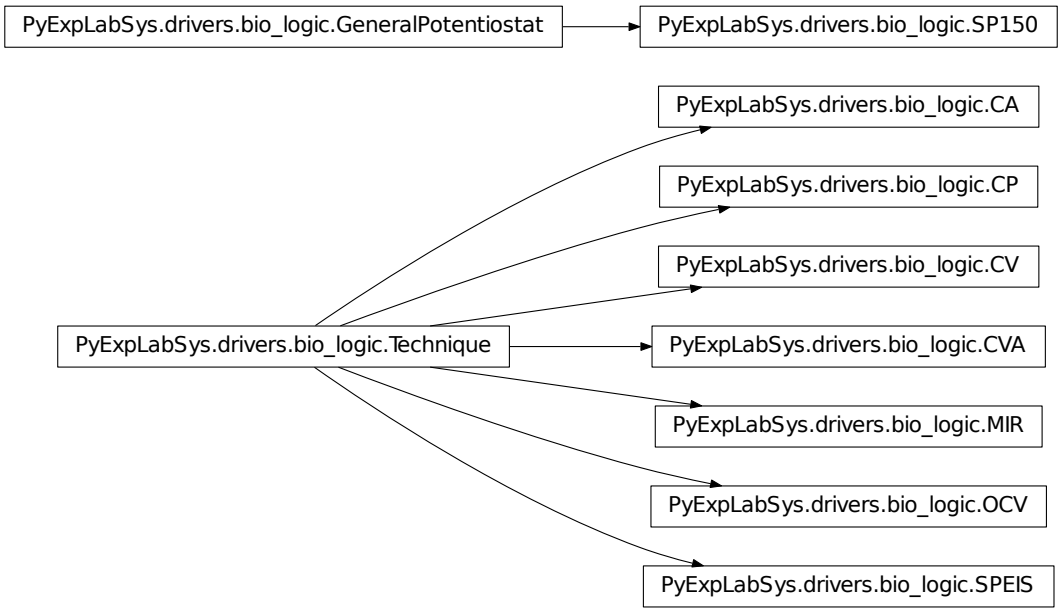
6.1.1 Implementation status and details

There is a whole range of potentiostats (VMP2/VMP3, BiStat, VSP, SP-50/SP-150, MGP2, HVP-803, SP-200, SP-300) that are covered by the same interface and which therefore with very small adjustments could be supported by this module as well. Currently only the SP-150 is implemented, because that is the only one we have in-house. See the section *Use/Implement a new potentiostat* for details.

This module currently implements the handful of techniques that are used locally (see a complete list at the top of the *module documentation*). It does however implement a `Technique` base class, which does all the hard work of formatting the technique arguments correctly, which means that writing a adding a new technique is limited to writing a new class in which it is only required to specify which input arguments the technique takes and which fields it outputs data.

Inheritance diagram

An inheritance diagram for the instruments and techniques (click the classes to get to their API documentation):



6.1.2 Usage Example

The example below is a complete run-able file demonstrating how to use the module, demonstrated with the OCV technique.

```
"""OCV example"""

from __future__ import print_function
import time
from PyExpLabSys.drivers.bio_logic import SP150, OCV


def run_ocv():
    """Test the OCV technique"""
    ip_address = '192.168.0.257'  # REPLACE THIS WITH A VALID IP
    # Instantiate the instrument and connect to it
    sp150 = SP150(ip_address)
```

(continues on next page)

(continued from previous page)

```
    sp150.connect()

    # Instantiate the technique. Make sure to give values for all the
    # arguments where the default values does not fit your purpose. The
    # default values can be viewed in the API documentation for the
    # technique.
    ocv = OCV(rest_time_T=0.2,
              record_every_dE=10.0,
              record_every_dT=0.01)

    # Load the technique onto channel 0 of the potentiostat and start it
    sp150.load_technique(0, ocv)
    sp150.start_channel(0)

    time.sleep(0.1)
    while True:
        # Get the currently available data on channel 0 (only what has
        # been gathered since last get_data)
        data_out = sp150.get_data(0)

        # If there is none, assume the technique has finished
        if data_out is None:
            break

        # The data is available in lists as attributes on the data
        # object. The available data fields are listed in the API
        # documentation for the technique.
        print("Time:", data_out.time)
        print("Ewe:", data_out.Ewe)

        # If numpy is installed, the data can also be retrieved as
        # numpy arrays
        #print('Time:', data_out.time_numpy)
        #print('Ewe:', data_out.Ewe_numpy)
        time.sleep(0.1)

    sp150.stop_channel(0)
    sp150.disconnect()

if __name__ == '__main__':
    run_ocv()
```

This example covers how most of the techniques would be used. A noticeable exception is the SPEIS technique, which returns data from two different processes, on two different sets of data fields. It will be necessary to take this into account, where the data is retrieved along these lines:

```
while True:
    time.sleep(0.1)
    data_out = sp150.get_data(0)
    if data_out is None:
        break

    print('Process index', data_out.process)
    if data_out.process == 0:
        print('time', data_out.time)
        print('Ewe', data_out.Ewe)
```

(continues on next page)

(continued from previous page)

```
        print('I', data_out.I)
        print('step', data_out.step)
    else:
        print('freq', data_out.freq)
        print('abs_Ewe', data_out.abs_Ewe)
        print('abs_I', data_out.abs_I)
        print('Phase_Zwe', data_out.Phase_Zwe)
        print('Ewe', data_out.Ewe)
        print('I', data_out.I)
        print('abs_Ece', data_out.abs_Ece)
        print('abs_Ice', data_out.abs_Ice)
        print('Phase_Zce', data_out.Phase_Zce)
        print('Ece', data_out.Ece)
        # Note, no time datafield, but a t
        print('t', data_out.t)
        print('Irange', data_out.Irange)
        print('step', data_out.step)
```

For more examples of how to use the other techniques, see the integration test file on Github

External documentation

The full documentation for the EC-Lab Development Package is available from the BioLogic website BioLogic Website and locally at CINF on the wiki.

Use/Implement a new potentiostat

To use a potentiostat that has not already been implemented, there are basically two options; *implement it* (3 lines of code excluding documentation) or *use the GeneralPotentiostat class directly*.

6.1.3 Implement a new potentiostat

The SP150 class is implemented in the following manner:

```
class SP150(GeneralPotentiostat):
    """Specific driver for the SP-150 potentiostat"""

    def __init__(self, address, EClib_dll_path=None):
        """Initialize the SP150 potentiostat driver

        See the __init__ method for the GeneralPotentiostat class for an
        explanation of the arguments.
        """
        super(SP150, self).__init__(
            type_='KBIO_DEV_SP150',
            address=address,
            EClib_dll_path=EClib_dll_path
        )
```

As it can be seen, the implementation of a new potentiostat boils down to:

- Inherit from `GeneralPotentiostat`
- Take `address` and `EClib_dll_path` as arguments to `__init__`

- Call `__init__` from *GeneralPotentiostat* with the potentiostat type string and forward the `address` and `EClib_dll_path`. The complete list of potentiostat type strings are listed in *DEVICE_CODES*.

6.1.4 Use `GeneralPotentionstat`

As explained in *Implement a new potentiostat*, the only thing that is required to use a new potentiostat is to call *GeneralPotentiostat* with the appropriate potentiostat type string. As an alternative to implementing the potentiostat in the module, this can of course also be done directly. This example shows e.g. how to get a driver for the BiStat potentiostat:

```
from PyExpLabSys.drivers.bio_logic import GeneralPotentiostat
potentiostat = GeneralPotentiostat(
    type_='KBIO_DEV_BISTAT',
    address='192.168.0.257',  # Replace this with a valid IP ;)
    EClib_dll_path=None
)
```

The complete list of potentiostat type strings are listed in *DEVICE_CODES*.

Use/Implement a new technique

To use a new technique, it will be required to implement it as a new class. This can of course both be done directly in the module and contributed back upstream or in custom code. The implementation of the OCV technique looks as follows:

```
class OCV(Technique):
    """Open Circuit Voltage (OCV) technique class.

    The OCV technique returns data on fields (in order):

    * time (float)
    * Ewe (float)
    * Ece (float) (only wmp3 series hardware)
    """

    #: Data fields definition
    data_fields = {
        'vmp3': [DataField('Ewe', c_float), DataField('Ece', c_float)],
        'sp300': [DataField('Ewe', c_float)],
    }

    def __init__(self, rest_time_T=10.0, record_every_dE=10.0,
                 record_every_dT=0.1, E_range='KBIO_ERANGE_AUTO'):
        """Initialize the OCV technique

        Args:
            rest_time_t (float): The amount of time to rest (s)
            record_every_dE (float): Record every dE (V)
            record_every_dT  (float): Record evergy dT (s)
            E_range (str): A string describing the E range to use, see the
                :data:`E_RANGES` module variable for possible values
        """
        args = (
            TechnArg('Rest_time_T', 'single', rest_time_T, '>=', 0),
            TechnArg('Record_every_dE', 'single', record_every_dE, '>=', 0),
```

(continues on next page)

(continued from previous page)

```
            TechnArg('Record_every_dT', 'single', record_every_dT, '>=', 0),
            TechnArg('E_Range', E_RANGES, E_range, 'in', E_RANGES.values()),
        )
        super(OCV, self).__init__(args, 'ocv.ecc')
```

As it can be seen, the new technique must inherit from *Technique*. This base class is responsible for bounds checking of the arguments and for formatting them in the appropriate way before sending them to the potentiostat.

A class variable with a dict named `data_fields` must be defined, that describes which data fields the technique makes data available at. See the docstring for *Technique* for a complete description of what the contents must be.

In the `__init__` method, the technique implementation must reflect all the arguments the *specification* lists for the technique (in this module, these arguments are made more Pythonic by; changing the names to follow naming conventions except that symbols are still capital, infer the number of arguments in lists instead of specifically asking for them and by leaving out arguments that can only have one value). All of the arguments from the *specification* must then be put, in order, into the args tuple in the form the *TechniqueArgument* instances. The specification for the arguments for the *TechniqueArgument* is in its docstring.

Then, finally, *Technique.__init__()* is called via super, with the args and the technique filename (is listed in the *specification*) as arguments.

The last thing to do is to add an entry for the technique in the *TECHNIQUE_IDENTIFIERS_TO_CLASS* dict, to indicate where the instrument should look, to figure out what the data layout is, when it receives data from this technique. If the new technique is implemented in stand alone code, this will need to be hacked (see the attached example).

In `this file` is a complete (re)implementation of the OCV technique as it would look if it was developed outside of the module.

6.1.5 bio_logic API

This module is a Python implementation of a driver around the EC-lib DLL. It can be used to control at least the SP-150 potentiostat from Bio-Logic under 32 bit Windows.

Note: If it is desired to run this driver and the EC-lab development DLL on **Linux**, this can be **achieved with Wine**. This will require installing both the EC-lab development package AND Python inside Wine. Getting Python installed is easiest, if it is a 32 bit Wine environment, so before starting, it is recommended to set such an environment up. **NOTE:** In a cursory test, it appears that also EClab itself runs under Wine.

Note: When using the different techniques with the EC-lib DLL, different technique files must be passed to the library, depending on **which series the instrument is in (VMPW series or SP-300 series)**. However, the definition of which instruments are in which series was not clear from the specification, so instead it was copied from one of the examples. The definition used is that if the device id of your instrument (see *DEVICE_CODES* for the full list of device ids) is in the *SP300SERIES* list, then it is regarded as a SP-300 series device. If problems are encountered when loading the technique, then this might be the issues and it will posible be necessary to customize *SP300SERIES*.

Note: On **64-bit Windows systems**, you should use the `EClib64.dll` instead of the `EClib.dll`. If the EC-lab development package is installed in the default location, this driver will try and load the correct DLL automatically, if not, the DLL path will need to passed explicitely and the user will need to take 32 vs. 64 bit into account. **NOTE:** The relevant 32 vs. 64 bit status is that of Windows, not of Python.

Note: All methods mentioned in the documentation are implemented unless mentioned in the list below:

- (General) BL_GetVolumeSerialNumber (Not implemented)
- (Communications) BL_TestCommSpeed (Not implemented)
- (Communications) BL_GetUSBdeviceinfos (Not implemented)
- (Channel information) BL_GetHardConf (N/A, only available w. SP300 series)
- (Channel information) BL_SetHardConf (N/A, only available w. SP300 series)
- (Technique) BL_UpdateParameters (Not implemented)
- (Start stop) BL_StartChannels (Not implemented)
- (Start stop) BL_StopChannels (Not implemented)
- (Data) BL_GetFCTData (Not implemented)
- (Misc) BL_SetExperimentInfos (Not implemented)
- (Misc) BL_GetExperimentInfos (Not implemented)
- (Misc) BL_SendMsg (Not implemented)
- (Misc) BL_LoadFlash (Not implemented)

Instrument classes:

- *GeneralPotentiostat*
- *SP150*

Techniques:

- *CA*
- *CP*
- *CV*
- *CVA*
- *MIR*
- *OCV*
- *SPEIS*
- *Technique*

class PyExpLabSys.drivers.bio_logic.**DataField**(*name*, *type*)

Bases: tuple

A named tuple used to defined a return data field for a technique

_asdict()

Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)

Make a new DataField object from a sequence or iterable

_replace(***kwds*)

Return a new DataField object replacing specified fields with new values

name
Alias for field number 0

type
Alias for field number 1

class PyExpLabSys.drivers.bio_logic.**TechniqueArgument**(*label*, *type*, *value*, *check*, *check_argument*)

Bases: tuple

The TechniqueArgument instance, that are used as args arguments, are named tuples with the following fields (in order):

- label (str): the argument label mentioned in the *specification*
- type (str): the type used in the *specification* ('bool', 'single' and 'integer') and possibly wrap [] around to indicate an array e.g. [bool]`
- value: The value to be passed, will usually be forwarded from __init__ args
- check (str): The bounds check to perform (if any), possible values are '>=', 'in' and 'in_float_range'
- check_argument: The argument(s) for the bounds check. For 'in' should be a float or int, for 'in' should be a sequence and for 'in_float_range' should be a tuple of two floats

_asdict()
Return a new OrderedDict which maps field names to their values

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new TechniqueArgument object from a sequence or iterable

_replace(***kwds*)
Return a new TechniqueArgument object replacing specified fields with new values

check
Alias for field number 3

check_argument
Alias for field number 4

label
Alias for field number 0

type
Alias for field number 1

value
Alias for field number 2

class PyExpLabSys.drivers.bio_logic.**GeneralPotentiostat**(*type_*, *address*, *EClib_dll_path*)

Bases: object

General driver for the potentiostats that can be controlled by the EC-lib DLL

A driver for a specific potentiostat type will inherit from this class.

Raises ***ECLibError*** – All regular methods in this class use the EC-lib DLL communications library to talk with the equipment and they will raise this exception if this library reports an error. It will not be explicitly mentioned in every single method.

__init__(*type_*, *address*, *EClib_dll_path*)
Initialize the potentiostat driver

Parameters

- **type** (*str*) – The device type e.g. 'KBIO_DEV_SP150'
- **address** (*str*) – The address of the instrument, either IP address or USB0, USB1 etc
- **EClib_dll_path** (*str*) – The path to the EClib DLL. The default directory of the DLL is C:EC-Lab Development PackageEC-Lab Development Packageand the filename is either EClib64.dll or EClib.dll depending on whether the operating system is 64 of 32 Windows respectively. If no value is given the default location will be used and the 32/64 bit status inferred.

Raises **WindowsError** – If the EClib DLL cannot be found

id_number
Return the device id as an int

device_info
Return the device information.

Returns

The device information as a dict or None if the device is not connected.

Return type dict or None

get_lib_version()
Return the version of the EClib communications library.

Returns The version string for the library

Return type str

get_error_message(*error_code*)
Return the error message corresponding to error_code

Parameters **error_code** (*int*) – The error number to translate

Returns The error message corresponding to error_code

Return type str

connect(*timeout=5*)
Connect to the instrument and return the device info.

Parameters **timeout** (*int*) – The connect timeout

Returns

The device information as a dict or None if the device is not connected.

Return type dict or None

Raises ***ECLibCustomException*** – If this class does not match the device type

disconnect()
Disconnect from the device

test_connection()
Test the connection

load_firmware(*channels*, *force_reload=False*)
Load the library firmware on the specified channels, if it is not already loaded

Parameters

- **channels** (*list*) – List with 1 integer per channel (usually 16), (0=False and 1=True), that indicates which channels the firmware should be loaded on. NOTE: The length of the list must correspond to the number of channels supported by the equipment, not the number of channels installed. In most cases it will be 16.
- **force_reload** (*bool*) – If True the firmware is forcefully reloaded, even if it was already loaded

Returns

List of integers indicating the success of loading the firmware on the specified channel. 0 is success and negative values are errors, whose error message can be retrieved with the get_error_message method.

Return type list

is_channel_plugged(*channel*)
Test if the selected channel is plugged.

Parameters **channel** (*int*) – Selected channel (0-15 on most devices)

Returns Whether the channel is plugged

Return type bool

get_channels_plugged()
Get information about which channels are plugged.

Returns A list of channel plugged statusses as booleans

Return type (list)

get_channel_infos(*channel*)
Get information about the specified channel.

Parameters **channel** (*int*) – Selected channel, zero based (0-15 on most devices)

Returns

Channel infos dict. The dict is created by conversion from *ChannelInfos* class (type ctypes.Structure). See the documentation for that class for a list of available dict items. Besides the items listed, there are extra items for all the original items whose value can be converted from an integer code to a string. The keys for those values are suffixed by (translated).

Return type dict

get_message(*channel*)
Return a message from the firmware of a channel

load_technique(*channel*, *technique*, *first=True*, *last=True*)
Load a technique on the specified channel

Parameters

- **channel** (*int*) – The number of the channel to load the technique onto
- **technique** (Technique) – The technique to load
- **first** (*bool*) – Whether this technique is the first technique
- **last** (*bool*) – Thether this technique is the last technique

Raises ***ECLibError*** – On errors from the EClib communications library

define_bool_parameter(*label*, *value*, *index*, *tecc_param*)
Defines a boolean TECCParam for a technique

This is a library convinience function to fill out the TECCParam struct in the correct way for a boolean value.

Parameters

- **label** (*str*) – The label of the parameter
- **value** (*bool*) – The boolean value for the parameter
- **index** (*int*) – The index of the parameter
- **tecc_param** (TECCParam) – An TECCParam struct

define_single_parameter(*label*, *value*, *index*, *tecc_param*)
Defines a single (float) TECCParam for a technique

This is a library convinience function to fill out the TECCParam struct in the correct way for a single (float) value.

Parameters

- **label** (*str*) – The label of the parameter
- **value** (*float*) – The float value for the parameter
- **index** (*int*) – The index of the parameter
- **tecc_param** (TECCParam) – An TECCParam struct

define_integer_parameter(*label*, *value*, *index*, *tecc_param*)
Defines an integer TECCParam for a technique

This is a library convinience function to fill out the TECCParam struct in the correct way for a integer value.

Parameters

- **label** (*str*) – The label of the parameter
- **value** (*int*) – The integer value for the parameter
- **index** (*int*) – The index of the parameter
- **tecc_param** (TECCParam) – An TECCParam struct

start_channel(*channel*)
Start the channel

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

stop_channel(*channel*)
Stop the channel

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

get_current_values(*channel*)
Get the current values for the spcified channel

Parameters **channel** (*int*) – The number of the channel (zero based)

Returns A dict of current values information

Return type dict

get_data(*channel*)
Get data for the specified channel

Parameters **channel** (*int*) – The number of the channel (zero based)

Returns

A *KBIOData* object or None if no data was available

Return type *KBIOData*

convert_numeric_into_single(*numeric*)
Convert a numeric (integer) into a float

The buffer used to get data out of the device consist only of uint32s (most likely to keep its layout simple). To transfer a float, the EClib library uses a trick, wherein the value of the float is saved as a uint32, by giving the uint32 the integer values, whose bit-representation corresponds to the float that it should describe. This function is used to convert the integer back to the corresponding float.

NOTE: This trick can also be performed with ctypes along the lines of: `c_float.from_buffer(c_uint32(numeric))`, but in this driver the library version is used.

Parameters **numeric** (*int*) – The integer that represents a float

Returns The float value

Return type float

check_eclib_return_code(*error_code*)
Check a ECLib return code and raise the appropriate exception

class PyExpLabSys.drivers.bio_logic.**SP150**(*address*, *EClib_dll_path=None*)
Bases: *PyExpLabSys.drivers.bio_logic.GeneralPotentiostat*

Specific driver for the SP-150 potentiostat

__init__(*address*, *EClib_dll_path=None*)
Initialize the SP150 potentiostat driver

See the __init__ method for the GeneralPotentiostat class for an explanation of the arguments.

class PyExpLabSys.drivers.bio_logic.**KBIOData**(*c_databuffer*, *c_data_infos*, *c_current_values*, *instrument*)
Bases: object

Class used to represent data obtained with a get_data call

The data can be obtained as lists of floats through attributes on this class. The time is always available through the 'time' attribute. The attribute names for the rest of the data, are the same as their names as listed in the field_names attribute. E.g:

- kbio_data.Ewe
- kbio_data.I

Provided that numpy is installed, the data can also be obtained as numpy arrays by appending '_numpy' to the attribute name. E.g:

- kbio_data.Ewe.numpy
- kbio_data.I_numpy

__init__(*c_databuffer*, *c_data_infos*, *c_current_values*, *instrument*)
Initialize the KBIOData object

Parameters

- **c_databuffer** (Array of ctypes.c_uint32) – ctypes array of c_uint32 used as the data buffer
- **c_data_infos** (*DataInfos*) – Data information structure
- **c_current_values** (*CurrentValues*) – Current values structure
- **instrument** (*GeneralPotentiostat*) – Instrument instance, should be an instance of a subclass of *GeneralPotentiostat*

Raises ***ECLibCustomException*** – Where the error codes indicate the following:

* -20000 means that the technique has no entry in *TECHNIQUE_IDENTIFIERS_TO_CLASS* * -20001 means that the technique class has no data_fields class variable * -20002 means that the data_fields class variables of the technique does not contain the right information

_init_data_fields(*instrument*)
Initialize the data fields property

_parse_data(*c_databuffer*, *timebase*, *instrument*)
Parse the data

Parameters **timebase** (*float*) – The timebase for the time calculation

See *__init__()* for information about remaining args

data_field_names
Return a list of extra data fields names (besides time)

class PyExpLabSys.drivers.bio_logic.**Technique**(*args*, *technique_filename*)
Bases: object

Base class for techniques

All specific technique classes inherits from this class.

Properties available on the object:

- technique_filename (str): The name of the technique filename
- args (tuple): Tuple containing the Python version of the parameters (see *__init__()* for details)
- c_args (array of *TECCParam*): The c-types array of *TECCParam*

A specific technique, that inherits from this class **must** overwrite the **data_fields** class variable. It describes what the form is, of the data that the technique can receive. The variable should be a dict on the following form:

- Some techniques, like *OCV*, have different data fields depending on the series of the instrument. In these cases the dict must contain both a 'wmp3' and a 'sp300' key.
- For cases where the instrument class distinction mentioned above does not exist, like e.g. for *CV*, one can simply define a 'common' key.
- All three cases above assume that the first field of the returned data is a specially formatted time field, which must not be listed directly.
- Some techniques, like e.g. *SPEIS* returns data for two different processes, one of which does not contain the time field (it is assumed that the process that contains time is 0 and the one that does not is 1). In this case there must be a 'common' and a 'no-time' key (see the implementation of *SPEIS* for details).

All of the entries in the dict must point to an list of *DataField* named tuples, where the two arguments are the name and the C type of the field (usually c_float or c_uint32). The list of fields must be in the order the data fields is specified in the *specification*.

__init__(*args*, *technique_filename*)
Initialize a technique

Parameters

- **args** (*tuple*) – Tuple of technique arguments as TechniqueArgument instances
- **technique_filename** (*str*) – The name of the technique filename.

Note: This must be the vmp3 series version i.e. name.ecc NOT name4.ecc, the replacement of technique file names are taken care of in load technique

c_args(*instrument*)
Return the arguments struct

Parameters **instrument** (*GeneralPotentiostat*) – Instrument instance, should be an instance of a subclass of *GeneralPotentiostat*

Returns An ctypes array of *TECCParam*

Return type array of *TECCParam*

Raises ***ECLibCustomException*** – Where the error codes indicate the following:

* -10000 means that an *TechniqueArgument* failed the 'in' test * -10001 means that an *TechniqueArgument* failed the '>=' test * -10002 means that an *TechniqueArgument* failed the 'in_float_range' test * -10010 means that it was not possible to find a conversion function for the defined type * -10011 means that the value cannot be converted with the conversion function

_init_c_args(*instrument*)
Initialize the arguments struct

Parameters **instrument** (*GeneralPotentiostat*) – Instrument instance, should be an instance of a subclass of *GeneralPotentiostat*

static _check_arg(*arg*)
Perform bounds check on a single argument

class PyExpLabSys.drivers.bio_logic.**OCV**(*rest_time_T=10.0*, *record_every_dE=10.0*, *record_every_dT=0.1*, *E_range='KBIO_ERANGE_AUTO'*)
Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Open Circuit Voltage (OCV) technique class.

The OCV technique returns data on fields (in order):

- time (float)
- Ewe (float)
- Ece (float) (only wmp3 series hardware)

data_fields = {'sp300': [DataField(name='Ewe', type=<class 'ctypes.c_float'>)], 'vmp3
Data fields definition

__init__(*rest_time_T=10.0*, *record_every_dE=10.0*, *record_every_dT=0.1*, *E_range='KBIO_ERANGE_AUTO'*)
Initialize the OCV technique

Parameters

- **rest_time_t** (*float*) – The amount of time to rest (s)

- **record_every_dE** (*float*) – Record every dE (V)
- **record_every_dT** (*float*) – Record evergy dT (s)
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values

class PyExpLabSys.drivers.bio_logic.**CV**(*vs_initial*, *voltage_step*, *scan_rate*, *record_every_dE=0.1*, *average_over_dE=True*, *N_cycles=0*, *begin_measuring_I=0.5*, *end_measuring_I=1.0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)

Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Cyclic Voltammetry (CV) technique class.

The CV technique returns data on fields (in order):

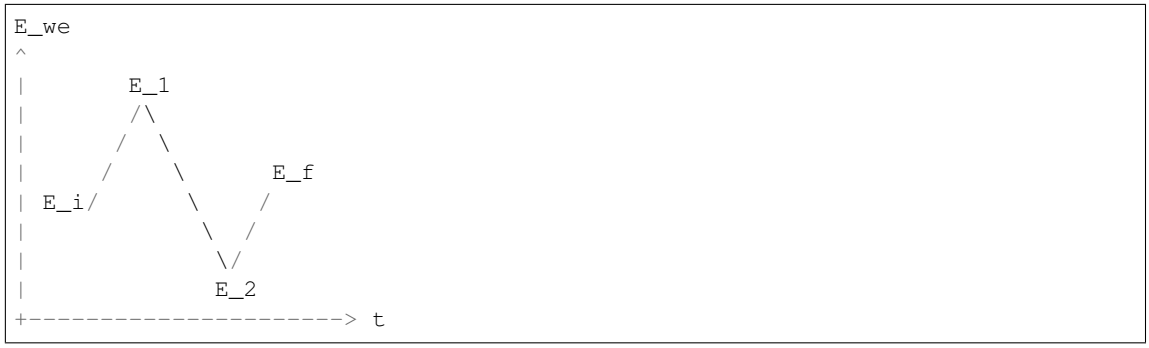
- time (float)
- Ec (float)
- I (float)
- Ewe (float)
- cycle (int)

data_fields = {'common': [DataField(name='Ec', type=<class 'ctypes.c_float'>), DataFi

Data fields definition

__init__(*vs_initial*, *voltage_step*, *scan_rate*, *record_every_dE=0.1*, *average_over_dE=True*, *N_cycles=0*, *begin_measuring_I=0.5*, *end_measuring_I=1.0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)

Initialize the CV technique:



Parameters

- **vs_initial** (*list*) – List (or tuple) of 5 booleans indicating whether the current step is vs. the initial one
- **voltage_step** (*list*) – List (or tuple) of 5 floats (Ei, E1, E2, Ei, Ef) indicating the voltage steps (V)
- **scan_rate** (*list*) – List (or tuple) of 5 floats indicating the scan rates (mV/s)
- **record_every_dE** (*float*) – Record every dE (V)
- **average_over_dE** (*bool*) – Whether averaging should be performed over dE

- **N_cycles** (*int*) – The number of cycles
- **begin_measuring_I** (*float*) – Begin step accumulation, 1 is 100%
- **end_measuring_I** (*float*) – Begin step accumulation, 1 is 100%
- **I_Range** (*str*) – A string describing the I range, see the *I_RANGES* module variable for possible values
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values
- **Bandwidth** (*str*) – A string describing the bandwidth setting, see the *BANDWIDTHS* module variable for possible values

Raises **ValueError** – If vs_initial, voltage_step and scan_rate are not all of length 5

class PyExpLabSys.drivers.bio_logic.**CVA**(*vs_initial_scan*, *voltage_scan*, *scan_rate*, *vs_initial_step*, *voltage_step*, *duration_step*, *record_every_dE=0.1*, *average_over_dE=True*, *N_cycles=0*, *begin_measuring_I=0.5*, *end_measuring_I=1.0*, *record_every_dT=0.1*, *record_every_dI=1*, *trig_on_off=False*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)

Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Cyclic Voltammetry Advanced (CVA) technique class.

The CVA technique returns data on fields (in order):

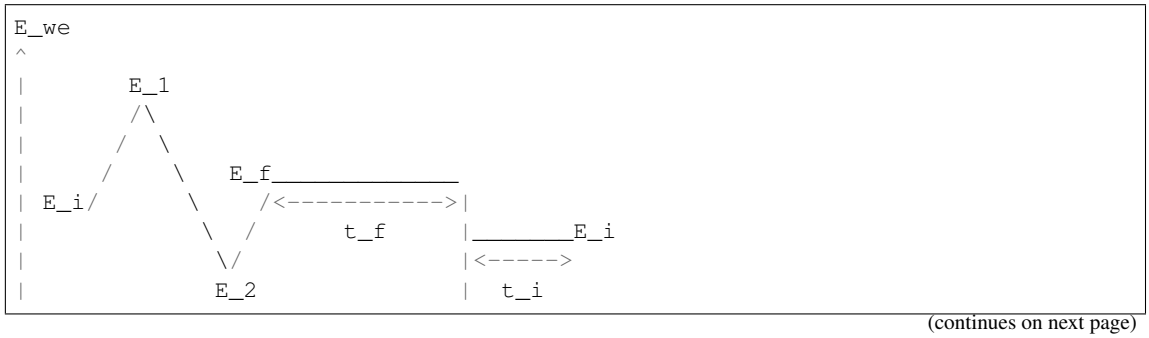
- time (float)
- Ec (float)
- I (float)
- Ewe (float)
- cycle (int)

data_fields = {'common': [DataField(name='Ec', type=<class 'ctypes.c_float'>), DataFi

Data fields definition

__init__(*vs_initial_scan*, *voltage_scan*, *scan_rate*, *vs_initial_step*, *voltage_step*, *duration_step*, *record_every_dE=0.1*, *average_over_dE=True*, *N_cycles=0*, *begin_measuring_I=0.5*, *end_measuring_I=1.0*, *record_every_dT=0.1*, *record_every_dI=1*, *trig_on_off=False*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)

Initialize the CVA technique:



(continues on next page)

(continued from previous page)

```
+-----------------------------+-------------> t
                              |
                           trigger
```

Parameters

- **vs_initial_scan** (*list*) – List (or tuple) of 4 booleans indicating whether the current scan is vs. the initial one
- **voltage_scan** (*list*) – List (or tuple) of 4 floats (Ei, E1, E2, Ef) indicating the voltage steps (V) (see diagram above)
- **scan_rate** (*list*) – List (or tuple) of 4 floats indicating the scan rates (mV/s)
- **record_every_dE** (*float*) – Record every dE (V)
- **average_over_dE** (*bool*) – Whether averaging should be performed over dE
- **N_cycles** (*int*) – The number of cycles
- **begin_measuring_I** (*float*) – Begin step accumulation, 1 is 100%
- **end_measuring_I** (*float*) – Begin step accumulation, 1 is 100%
- **vs_initial_step** (*list*) – A list (or tuple) of 2 booleans indicating whether this step is vs. the initial one
- **voltage_step** (*list*) – A list (or tuple) of 2 floats indicating the voltage steps (V)
- **duration_step** (*list*) – A list (or tuple) of 2 floats indicating the duration of each step (s)
- **record_every_dT** (*float*) – A float indicating the change in time that leads to a point being recorded (s)
- **record_every_dI** (*float*) – A float indicating the change in current that leads to a point being recorded (A)
- **trig_on_off** (*bool*) – A boolean indicating whether to use the trigger
- **I_Range** (*str*) – A string describing the I range, see the *I_RANGES* module variable for possible values
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values
- **Bandwidth** (*str*) – A string describing the bandwidth setting, see the *BANDWIDTHS* module variable for possible values

Raises **ValueError** – If vs_initial, voltage_step and scan_rate are not all of length 5

class PyExpLabSys.drivers.bio_logic.**CP**(*current_step=(5e-05,), vs_initial=(False,), duration_step=(10.0,), record_every_dT=0.1, record_every_dE=0.001, N_cycles=0, I_range='KBIO_IRANGE_100uA', E_range='KBIO_ERANGE_2_5', bandwidth='KBIO_BW_5'*)

Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Chrono-Potentiometry (CP) technique class.

The CP technique returns data on fields (in order):

- time (float)
- Ewe (float)
- I (float)
- cycle (int)

data_fields = {'common': [DataField(name='Ewe', type=<class 'ctypes.c_float'>), DataF
Data fields definition

__init__(*current_step=(5e-05,)*, *vs_initial=(False,)*, *duration_step=(10.0,)*, *record_every_dT=0.1*, *record_every_dE=0.001*, *N_cycles=0*, *I_range='KBIO_IRANGE_100uA'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)
Initialize the CP technique

NOTE: The current_step, vs_initial and duration_step must be a list or tuple with the same length.

Parameters

- **current_step** (*list*) – List (or tuple) of floats indicating the current steps (A). See NOTE above.
- **vs_initial** (*list*) – List (or tuple) of booleans indicating whether the current steps is vs. the initial one. See NOTE above.
- **duration_step** (*list*) – List (or tuple) of floats indicating the duration of each step (s). See NOTE above.
- **record_every_dT** (*float*) – Record every dT (s)
- **record_every_dE** (*float*) – Record every dE (V)
- **N_cycles** (*int*) – The number of times the technique is REPEATED. NOTE: This means that the default value is 0 which means that the technique will be run once.
- **I_Range** (*str*) – A string describing the I range, see the *I_RANGES* module variable for possible values
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values
- **Bandwidth** (*str*) – A string describing the bandwidth setting, see the *BANDWIDTHS* module variable for possible values

Raises **ValueError** – On bad lengths for the list arguments

class PyExpLabSys.drivers.bio_logic.**CA**(*voltage_step=(0.35,)*, *vs_initial=(False,)*, *duration_step=(10.0,)*, *record_every_dT=0.1*, *record_every_dI=5e-06*, *N_cycles=0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)
Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Chrono-Amperometry (CA) technique class.

The CA technique returns data on fields (in order):

- time (float)
- Ewe (float)
- I (float)
- cycle (int)

data_fields = {'common': [DataField(name='Ewe', type=<class 'ctypes.c_float'>), DataF
Data fields definition

__init__(*voltage_step=(0.35,)*, *vs_initial=(False,)*, *duration_step=(10.0,)*, *record_every_dT=0.1*, *record_every_dI=5e-06*, *N_cycles=0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)
Initialize the CA technique

NOTE: The voltage_step, vs_initial and duration_step must be a list or tuple with the same length.

Parameters

- **voltage_step** (*list*) – List (or tuple) of floats indicating the voltage steps (A). See NOTE above.
- **vs_initial** (*list*) – List (or tuple) of booleans indicating whether the current steps is vs. the initial one. See NOTE above.
- **duration_step** (*list*) – List (or tuple) of floats indicating the duration of each step (s). See NOTE above.
- **record_every_dT** (*float*) – Record every dT (s)
- **record_every_dI** (*float*) – Record every dI (A)
- **N_cycles** (*int*) – The number of times the technique is REPEATED. NOTE: This means that the default value is 0 which means that the technique will be run once.
- **I_Range** (*str*) – A string describing the I range, see the *I_RANGES* module variable for possible values
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values
- **Bandwidth** (*str*) – A string describing the bandwidth setting, see the *BANDWIDTHS* module variable for possible values

Raises **ValueError** – On bad lengths for the list arguments

class PyExpLabSys.drivers.bio_logic.**SPEIS**(*vs_initial*, *vs_final*, *initial_voltage_step*, *final_voltage_step*, *duration_step*, *step_number*, *record_every_dT=0.1*, *record_every_dI=5e-06*, *final_frequency=100000.0*, *initial_frequency=100.0*, *sweep=True*, *amplitude_voltage=0.1*, *frequency_number=1*, *average_n_times=1*, *correction=False*, *wait_for_steady=1.0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)
Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Staircase Potentio Electrochemical Impedance Spectroscopy (SPEIS) technique class

The SPEIS technique returns data with a different set of fields depending on which process steps it is in. If it is in process step 0 it returns data on the following fields (in order):

- time (float)
- Ewe (float)
- I (float)
- step (int)

If it is in process 1 it returns data on the following fields:

- freq (float)
- abs_Ewe (float)
- abs_I (float)
- Phase_Zwe (float)
- Ewe (float)
- I (float)
- abs_Ece (float)
- abs_Ice (float)
- Phase_Zce (float)
- Ece (float)
- t (float)
- Irange (float)
- step (float)

Which process it is in, can be checked with the `process` property on the *KBIOData* object.

data_fields = {'common': [DataField(name='Ewe', type=<class 'ctypes.c_float'>), DataF
Data fields definition

__init__(*vs_initial*, *vs_final*, *initial_voltage_step*, *final_voltage_step*, *duration_step*, *step_number*, *record_every_dT=0.1*, *record_every_dI=5e-06*, *final_frequency=100000.0*, *initial_frequency=100.0*, *sweep=True*, *amplitude_voltage=0.1*, *frequency_number=1*, *average_n_times=1*, *correction=False*, *wait_for_steady=1.0*, *I_range='KBIO_IRANGE_AUTO'*, *E_range='KBIO_ERANGE_2_5'*, *bandwidth='KBIO_BW_5'*)
Initialize the SPEIS technique

Parameters

- **vs_initial** (*bool*) – Whether the voltage step is vs. the initial one
- **vs_final** (*bool*) – Whether the voltage step is vs. the final one
- **initial_step_voltage** (*float*) – The initial step voltage (V)
- **final_step_voltage** (*float*) – The final step voltage (V)
- **duration_step** (*float*) – Duration of step (s)
- **step_number** (*int*) – The number of voltage steps
- **record_every_dT** (*float*) – Record every dT (s)
- **record_every_dI** (*float*) – Record every dI (A)
- **final_frequency** (*float*) – The final frequency (Hz)
- **initial_frequency** (*float*) – The initial frequency (Hz)
- **sweep** (*bool*) – Sweep linear/logarithmic (True for linear points spacing)
- **amplitude_voltage** (*float*) – Amplitude of sinus (V)
- **frequency_number** (*int*) – The number of frequencies
- **average_n_times** (*int*) – The number of repeat times used for frequency averaging

- **correction** (*bool*) – Non-stationary correction
- **wait_for_steady** (*float*) – The number of periods to wait before each frequency
- **I_Range** (*str*) – A string describing the I range, see the *I_RANGES* module variable for possible values
- **E_range** (*str*) – A string describing the E range to use, see the *E_RANGES* module variable for possible values
- **Bandwidth** (*str*) – A string describing the bandwidth setting, see the *BANDWIDTHS* module variable for possible values

Raises **ValueError** – On bad lengths for the list arguments

class PyExpLabSys.drivers.bio_logic.**MIR**(*rcmp_value*)
Bases: *PyExpLabSys.drivers.bio_logic.Technique*

Manual IR (MIR) technique class

The MIR technique returns no data.

data_fields = {}
Data fields definition

__init__(*rcmp_value*)
Initialize the MIR technique

Parameters **rcmp_value** (*float*) – The R value to compensate

class PyExpLabSys.drivers.bio_logic.**DeviceInfos**
Bases: _ctypes.Structure

Device information struct

Fields:

- DeviceCode <class 'ctypes.c_int'>
- RAMsize <class 'ctypes.c_int'>
- CPU <class 'ctypes.c_int'>
- NumberOfChannels <class 'ctypes.c_int'>
- NumberOfSlots <class 'ctypes.c_int'>
- FirmwareVersion <class 'ctypes.c_int'>
- FirmwareDate_yyyy <class 'ctypes.c_int'>
- FirmwareDate_mm <class 'ctypes.c_int'>
- FirmwareDate_dd <class 'ctypes.c_int'>
- HTdisplayOn <class 'ctypes.c_int'>
- NbOfConnectedPC <class 'ctypes.c_int'>

class PyExpLabSys.drivers.bio_logic.**ChannelInfos**
Bases: _ctypes.Structure

Channel information structure

Fields:

- Channel <class 'ctypes.c_int'>
- BoardVersion <class 'ctypes.c_int'>

- BoardSerialNumber <class 'ctypes.c_int'>
- FirmwareCode <class 'ctypes.c_int'>
- FirmwareVersion <class 'ctypes.c_int'>
- XilinxVersion <class 'ctypes.c_int'>
- AmpCode <class 'ctypes.c_int'>
- NbAmp <class 'ctypes.c_int'>
- LCboard <class 'ctypes.c_int'>
- Zboard <class 'ctypes.c_int'>
- MUXboard <class 'ctypes.c_int'>
- GPRAboard <class 'ctypes.c_int'>
- MemSize <class 'ctypes.c_int'>
- MemFilled <class 'ctypes.c_int'>
- State <class 'ctypes.c_int'>
- MaxIRange <class 'ctypes.c_int'>
- MinIRange <class 'ctypes.c_int'>
- MaxBandwidth <class 'ctypes.c_int'>
- NbOfTechniques <class 'ctypes.c_int'>

class PyExpLabSys.drivers.bio_logic.**CurrentValues**

Bases: _ctypes.Structure

Current values structure

Fields:

- State <class 'ctypes.c_int'>
- MemFilled <class 'ctypes.c_int'>
- TimeBase <class 'ctypes.c_float'>
- Ewe <class 'ctypes.c_float'>
- EweRangeMin <class 'ctypes.c_float'>
- EweRangeMax <class 'ctypes.c_float'>
- Ece <class 'ctypes.c_float'>
- EceRangeMin <class 'ctypes.c_float'>
- EceRangeMax <class 'ctypes.c_float'>
- Eoverflow <class 'ctypes.c_int'>
- I <class 'ctypes.c_float'>
- IRange <class 'ctypes.c_int'>
- Ioverflow <class 'ctypes.c_int'>
- ElapsedTime <class 'ctypes.c_float'>
- Freq <class 'ctypes.c_float'>

- Rcomp <class 'ctypes.c_float'>
- Saturation <class 'ctypes.c_int'>

class PyExpLabSys.drivers.bio_logic.**DataInfos**
Bases: _ctypes.Structure

DataInfos structure

Fields:

- IRQskipped <class 'ctypes.c_int'>
- NbRaws <class 'ctypes.c_int'>
- NbCols <class 'ctypes.c_int'>
- TechniqueIndex <class 'ctypes.c_int'>
- TechniqueID <class 'ctypes.c_int'>
- ProcessIndex <class 'ctypes.c_int'>
- loop <class 'ctypes.c_int'>
- StartTime <class 'ctypes.c_double'>

class PyExpLabSys.drivers.bio_logic.**TECCParam**
Bases: _ctypes.Structure

Technique parameter

Fields:

- ParamStr <class 'PyExpLabSys.drivers.bio_logic.c_char_Array_64'>
- ParamType <class 'ctypes.c_int'>
- ParamVal <class 'ctypes.c_int'>
- ParamIndex <class 'ctypes.c_int'>

class PyExpLabSys.drivers.bio_logic.**TECCParams**
Bases: _ctypes.Structure

Technique parameters

Fields:

- len <class 'ctypes.c_int'>
- pParams <class 'PyExpLabSys.drivers.bio_logic.LP_TECCParam'>

exception PyExpLabSys.drivers.bio_logic.**ECLibException**(*message*, *error_code*)
Bases: exceptions.Exception

Base exception for all ECLib exceptions

__init__(*message*, *error_code*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception PyExpLabSys.drivers.bio_logic.**ECLibError**(*message*, *error_code*)
Bases: *PyExpLabSys.drivers.bio_logic.ECLibException*

Exception for ECLib errors

__init__(*message*, *error_code*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception PyExpLabSys.drivers.bio_logic.**ECLibCustomException**(*message*, *error_code*)

Bases: *PyExpLabSys.drivers.bio_logic.ECLibException*

Exceptions that does not originate from the lib

__init__(*message*, *error_code*)
x.__init__(...) initializes x; see help(type(x)) for signature

PyExpLabSys.drivers.bio_logic.**structure_to_dict**(*structure*)
Convert a ctypes.Structure to a dict

PyExpLabSys.drivers.bio_logic.**reverse_dict**(*dict_*)
Reverse the key/value status of a dict

PyExpLabSys.drivers.bio_logic.**DEVICE_CODES = {0: 'KBIO_DEV_VMP', 1: 'KBIO_DEV_VMP2', 2:**
Device number to device name translation dict

PyExpLabSys.drivers.bio_logic.**FIRMWARE_CODES = {0: 'KBIO_FIRM_NONE', 1: 'KBIO_FIRM_INTERI**
Firmware number to firmware name translation dict

PyExpLabSys.drivers.bio_logic.**AMP_CODES = {0: 'KBIO_AMPL_NONE', 1: 'KBIO_AMPL_2A', 2: 'I**
Amplifier number to aplifier name translation dict

PyExpLabSys.drivers.bio_logic.**I_RANGES = {0: 'KBIO_IRANGE_100pA', 1: 'KBIO_IRANGE_1nA', 2**
I range number to I range name translation dict

PyExpLabSys.drivers.bio_logic.**BANDWIDTHS = {1: 'KBIO_BW_1', 2: 'KBIO_BW_2', 3: 'KBIO_BW_**
Bandwidth number to bandwidth name translation dict

PyExpLabSys.drivers.bio_logic.**E_RANGES = {0: 'KBIO_ERANGE_2_5', 1: 'KBIO_ERANGE_5', 2:**
E range number to E range name translation dict

PyExpLabSys.drivers.bio_logic.**STATES = {0: 'KBIO_STATE_STOP', 1: 'KBIO_STATE_RUN', 2: 'I**
State number to state name translation dict

PyExpLabSys.drivers.bio_logic.**TECHNIQUE_IDENTIFIERS = {0: 'KBIO_TECHID_NONE', 100: 'KBIO_**
Technique number to technique name translation dict

PyExpLabSys.drivers.bio_logic.**TECHNIQUE_IDENTIFIERS_TO_CLASS = {'KBIO_TECHID_CA': <class 'I**
Technique name to technique class translation dict. IMPORTANT. Add newly implemented techniques to this dictionary

PyExpLabSys.drivers.bio_logic.**SP300SERIES = ['KBIO_DEV_SP100', 'KBIO_DEV_SP200', 'KBIO_DEV_**
List of devices in the WMP4/SP300 series

6.2 The 4d Systems module

6.2.1 Picaso Common

The 4d Systems module at present contains the Picaso Common driver, which at a minimum works for the Picaso uLCD-28PTU LCD display, but likely will also work for other displays in the same series.

Usage Example

```
import time
from PyExpLabSys.drivers.four_d_systems import PicasouLCD28PTU

# Text example
picaso = PicasouLCD28PTU(serial_device='/dev/ttyUSB0', baudrate=9600)
picaso.clear_screen()
for index in range(5):
    picaso.move_cursor(index, index)
    picaso.put_string('CINF')

# Touch example
picaso.move_cursor(7, 0)
picaso.put_string('Try and touch me!')
picaso.touch_set('enable')
for _ in range(25):
    time.sleep(0.2)
    print picaso.touch_get_status()
    print picaso.touch_get_coordinates()

picaso.close()
```

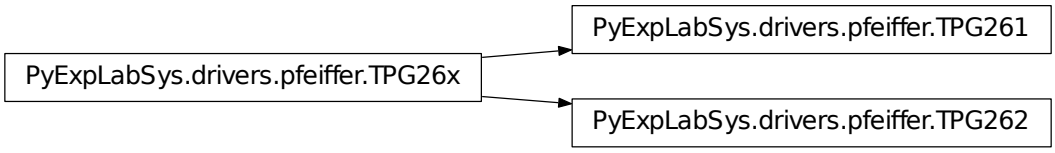
four_d_systems module

6.3 The pfeiffer module

The pfeiffer module contains drivers for equipment from Pfeiffer Vacuum. At present the module contains drivers for the *TPG 261* and *TPG 262* pressure measurement and control units.

6.3.1 TPG 26x

The TPG 261 and TPG 262 has the same communications protocol and therefore the driver has been implemented as a common driver in the *TPG26x* class, which the *TPG261* and *TPG262* classes inherit from, as illustrated below.



The driver implements only a sub set of the specification, but given that the ground work has already been done, it should be simple to implement more methods as they are needed.

Usage Example

The driver classes can be instantiated by specifying just the address of the serial communications port the unit is connected to:

```
from PyExpLabSys.drivers.pfeiffer import TPG262
tpg = TPG262(port='/dev/ttyUSB0')
value, (status_code, status_string) = tpg.pressure_gauge(1)
# or
value, _ = tpg.pressure_gauge(1)
unit = tpg.pressure_unit()
print 'pressure is {} {}'.format(value, unit)
```

If the baud rate on the TPG 26x unit has been changed away from the default setting of 9600, then the correct baud rate will need to be given as a parameter.

pfeiffer module

This module contains drivers for the following equipment from Pfeiffer Vacuum:

- **TPG 262 and TPG 261 Dual Gauge. Dual-Channel Measurement and Control** Unit for Compact Gauges

class PyExpLabSys.drivers.pfeiffer.**TPG26x**(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Bases: object

Abstract class that implements the common driver for the TPG 261 and TPG 262 dual channel measurement and control unit. The driver implements the following 6 commands out the 39 in the specification:

- PNR: Program number (firmware version)
- PR[1,2]: Pressure measurement (measurement data) gauge [1, 2]
- PRX: Pressure measurement (measurement data) gauge 1 and 2
- TID: Transmitter identification (gauge identification)
- UNI: Pressure unit
- RST: RS232 test

This class also contains the following class variables, for the specific characters that are used in the communication:

Variables

- **ETX** – End text (Ctrl-c), chr(3), \x15
- **CR** – Carriage return, chr(13), \r
- **LF** – Line feed, chr(10), \n
- **ENQ** – Enquiry, chr(5), \x05
- **ACK** – Acknowledge, chr(6), \x06
- **NAK** – Negative acknowledge, chr(21), \x15

__init__(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Initialize internal variables and serial connection

Parameters

- **port** (*str or int*) – The COM port to open. See the documentation for pyserial for an explanation of the possible value. The default value is '/dev/ttyUSB0'.
- **baudrate** (*int*) – 9600, 19200, 38400 where 9600 is the default

program_number()
Return the firmware version

Returns the firmware version

Return type str

pressure_gauge(*gauge=1*)
Return the pressure measured by gauge X

Parameters **gauge** (*int*) – The gauge number, 1 or 2

Raises **ValueError** – if gauge is not 1 or 2

Returns (value, (status_code, status_message))

Return type tuple

pressure_gauges()
Return the pressures measured by the gauges

Returns (value1, (status_code1, status_message1), value2, (status_code2, status_message2))

Return type tuple

gauge_identification()
Return the gauge identication

Returns (id_code_1, id_1, id_code_2, id_2)

Return type tuple

pressure_unit()
Return the pressure unit

Returns the pressure unit

Return type str

rs232_communication_test()
RS232 communication test

Returns the status of the communication test

Return type bool

class PyExpLabSys.drivers.pfeiffer.**TPG262**(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Bases: *PyExpLabSys.drivers.pfeiffer.TPG26x*

Driver for the TPG 262 dual channel measurement and control unit

__init__(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Initialize internal variables and serial connection

Parameters

- **port** (*str or int*) – The COM port to open. See the documentation for pyserial for an explanation of the possible value. The default value is '/dev/ttyUSB0'.
- **baudrate** (*int*) – 9600, 19200, 38400 where 9600 is the default

class PyExpLabSys.drivers.pfeiffer.**TPG261**(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Bases: *PyExpLabSys.drivers.pfeiffer.TPG26x*

Driver for the TPG 261 dual channel measurement and control unit

__init__(*port='/dev/ttyUSB0'*, *baudrate=9600*)
Initialize internal variables and serial connection

Parameters

- **port** (*str or int*) – The COM port to open. See the documentation for pyserial for an explanation of the possible value. The default value is '/dev/ttyUSB0'.
- **baudrate** (*int*) – 9600, 19200, 38400 where 9600 is the default

CHAPTER 7

Hardware Drivers Autogenerated Docs Only

This section documents the hardware drivers developed at CINF. Most of the drivers are for equipment to surface science such as mass spectrometers and pressure gauges, but there are also some drivers for more general equipment like temperature read out units.

The drivers in this section only has autogenerated API documentation.

7.1 The NGC2D module

7.1.1 Autogenerated API documentation for NGC2D

7.2 The agilent_34410A module

7.2.1 Autogenerated API documentation for agilent_34410A

Driver class for Agilent 34410A DMM

class PyExpLabSys.drivers.agilent_34410A.**Agilent34410ADriver**(*interface='lan'*, *hostname=''*, *connection_string=''*)

Bases: *PyExpLabSys.drivers.scpi.SCPI*

Driver for Agilent 34410A DMM

__init__(*interface='lan'*, *hostname=''*, *connection_string=''*)
x.__init__(...) initializes x; see help(type(x)) for signature

config_current_measurement()
Configures the instrument to measure current.

config_resistance_measurement()
Configures the instrument to measure resistance.

select_measurement_function(*function*)
Select a measurement function.

Keyword arguments: Function – A string stating the wanted measurement function.

read_configuration()
Read device configuration

set_auto_input_z(*auto=False*)
Change internal resitance

read()
Read a value from the device

PyExpLabSys.drivers.agilent_34410A.**main**()
Main function

7.3 The agilent_34972A module

7.3.1 Autogenerated API documentation for agilent_34972A

Driver class for Agilent 34972A multiplexer

class PyExpLabSys.drivers.agilent_34972A.**Agilent34972ADriver**(*interface='lan'*, *hostname=''*, *connection_string=''*)
Bases: *PyExpLabSys.drivers.scpi.SCPI*

Driver for Agilent 34972A multiplexer

__init__(*interface='lan'*, *hostname=''*, *connection_string=''*)
x.__init__(...) initializes x; see help(type(x)) for signature

read_single_scan()
Read a single scan-line

abort_scan()
Abort the scan

read_configuration()
Read device configuration

set_scan_interval(*interval*)
Set the scan interval

set_integration_time(*channel*, *nplc*)
Set integration time

read_scan_interval()
Read the scan interval

read_scan_list()
Return the scan list

set_scan_list(*channels*)
Set the scan list

7.4 The analogdevices_ad5667 module

7.4.1 Autogenerated API documentation for analogdevices_ad5667

Driver for the Analog Devices AD5667 2 channel analog output DAC

Implemented from the manual located here and the examples located here.

class PyExpLabSys.drivers.analogdevices_ad5667.**AD5667**
Driver for the Analog Devices AD5667 2 channel analog output DAC

__init__()
Initialize object properties

write_to_and_update_dac(*dac*, *value*)
Set a voltage value on the DAC

Parameters

- **dac** (*str*) – The name of the DAC to set. 'A', 'B' or 'both'
- **value** (*float*) – A float between 0.0 and 5.0

Raises **ValueError** – On bad DAC name or bad value

set_channel_A(*voltage*)
Set a voltage of channel A

Parameters **value** (*float*) – A float between 0.0 and 5.0

See *write_to_and_update_dac()* for details on exceptions

set_channel_B(*voltage*)
Set a voltage of channel B

Parameters **value** (*float*) – A float between 0.0 and 5.0

See *write_to_and_update_dac()* for details

set_both(*voltage*)
Set a voltage of both channels

Parameters **value** (*float*) – A float between 0.0 and 5.0

See *write_to_and_update_dac()* for details

PyExpLabSys.drivers.analogdevices_ad5667.**module_test**()
Simple module test

7.5 The bronkhorst module

7.5.1 Autogenerated API documentation for bronkhorst

Driver for Bronkhorst flow controllers, including simple test case

class PyExpLabSys.drivers.bronkhorst.**Bronkhorst**(*port*, *max_flow*)
Bases: object

Driver for Bronkhorst flow controllers

__init__(*port*, *max_flow*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Send commands to device and recieve reply

read_setpoint()
Read the current setpoint

read_flow()
Read the actual flow

set_flow(*setpoint*)
Set the desired setpoint, which could be a pressure

read_counter_value()
Read valve counter. Not fully implemented

set_control_mode()
Set the control mode to accept rs232 setpoint

read_serial()
Read the serial number of device

read_unit()
Read the flow unit

read_capacity()
Read ?? from device

7.6 The brooks_s_protocol module

7.6.1 Autogenerated API documentation for brooks_s_protocol

Driver for Brooks s-protocol

class PyExpLabSys.drivers.brooks_s_protocol.**Brooks**(*device*, *port='/dev/ttyUSB0'*)
Bases: object

Driver for Brooks s-protocol

__init__(*device*, *port='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

pack(*input_string*)
Turns a string in packed-ascii format

crc(*command*)
Calculate crc value of command

comm(*command*)
Implements low-level details of the s-protocol

read_flow()
Read the current flow-rate

read_full_range()
Report the full range of the device Apparantly this does not work for SLA-series...

set_flow(*flowrate*)
Set the setpoint of the flow

7.7 The cpx400dp module

7.7.1 Autogenerated API documentation for cpx400dp

Driver for CPX400DP power supply

exception PyExpLabSys.drivers.cpx400dp.**InterfaceOutOfBoundsError**(*value*)
Bases: exceptions.Exception

Error class for CPX400DP Driver

__init__(*value*)
x.__init__(...) initializes x; see help(type(x)) for signature

class PyExpLabSys.drivers.cpx400dp.**CPX400DPDriver**(*output*, *interface*, *hostname=''*, *device=''*, *tcp_port=0*)
Bases: *PyExpLabSys.drivers.scpi.SCPI*

Actual driver for the CPX400DP

__init__(*output*, *interface*, *hostname=''*, *device=''*, *tcp_port=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

set_voltage(*value*)
Sets the voltage

set_current_limit(*value*)
Sets the current limit

read_set_voltage()
Reads the set voltage

read_current_limit()
Reads the current limit

read_configuration_mode()
Return the depency mode between the channels

set_dual_output(*dual_output=True*)
Sets voltage tracking or dual output If dual_output is True, Dual output will be activated. If dual_output is False, Voltage tracking will be enabled

read_actual_voltage()
Reads the actual output voltage

read_actual_current()
Reads the actual output current

set_voltage_stepsize(*value*)
Sets the voltage step size

set_current_stepsize(*value*)
Sets the current step size

read_voltage_stepsize()
Reads the voltage step size

read_current_stepsize()
Read the current stepszie

increase_voltage()
Increase voltage one step

output_status(*output_on=False*)
Set the output status

read_output_status()
Read the output status

get_lock()
Lock the instrument for remote operation

7.8 The crowcon module

7.8.1 Autogenerated API documentation for crowcon

This module contains a driver for the Vortex gas alarm central

See also:

Docs for this implementation are on the wiki at: https://cinfwiki.fysik.dtu.dk/cinfwiki/Equipment#Vortex_Gas_Alarm_System or online at: http://www.crowcon.com/uk/products/control-panels/vortex.html

class PyExpLabSys.drivers.crowcon.**Status**(*code*, *value*)
Bases: tuple

code
Alias for field number 0

value
Alias for field number 1

PyExpLabSys.drivers.crowcon.**DetConfMap**
alias of PyExpLabSys.drivers.crowcon.DetectorConfigurationMap

PyExpLabSys.drivers.crowcon.**DetLev**
alias of PyExpLabSys.drivers.crowcon.DetectorLevels

PyExpLabSys.drivers.crowcon.**register_to_bool**(*register*)
Convert a register value to a boolean

0 is considered False, 65535 True and remaining integer values are invalid.

Parameters **register** (*int*) – The register value

Returns The boolean value

Return type bool

class PyExpLabSys.drivers.crowcon.**Vortex**(*serial_device*, *slave_address*, *debug=False*, *cache=True*, *retries=3*)

Bases: minimalmodbus.Instrument

Driver for the Vortex gas alarm central

Note: In the manual the register numbers are 1-based, but when sent to minimal modbus they need to be 0 based.

__init__(*serial_device*, *slave_address*, *debug=False*, *cache=True*, *retries=3*)

Initialize the driver

Parameters

- **serial_device** (*str*) – The serial device to use
- **slave_address** (*int*) – The address of the slave device
- **debug** (*bool*) – Whether debugging output from minimal modbus should be enabled
- **cache** (*bool*) – Whether system configuration values (which are expected not to change within the runtime of the program) should be cached

close()

Close the serial communication connection

read_register(**args*, ***kwargs*)

Read register from instrument (with retries)

The argument definition is the same as for the minimalmodbus method, see the full documentation for read_register for details.

read_string(**args*, ***kwargs*)

Read string from instrument (with retries)

The argument definition is the same as for the minimalmodbus method, see the full documentation read_string for details.

read_bool(*register*)

Read int from register and convert to boolean value

0 is considered False, 65535 True and remaining integer values are invalid.

Parameters **register** (*int*) – The register to read from

Returns The boolean value

Return type bool

get_type()

Get the device type

Returns The type of the device e.g. 'Vortex'

Return type str

get_system_status()

Get the system status

Returns

['All OK'] if no status bits (section 5.1.1 in the manual) has been set, otherwise one string for each of the status bits that was set.

Return type list

get_system_power_status()
Get the system power status

Returns A Status named tuple containing status code and string

Return type *Status*

get_serial_number()
Get the serial number

Returns The serial number

Return type str

get_system_name()
Get the serial number

Returns The system name

Return type str

get_number_installed_detectors()
Get the number of installed detector inputs

This value is cached if requested. See docstring for *__init__()*.

Returns The number of installed detectors

Return type int

get_number_installed_digital_outputs()
Get the number of installed digital outputs

Returns The number of installed digital inputs

Return type int

detector_configuration(*detector_number*)
Read detector configuration

This value is cached if requested. See docstring for *__init__()*.

Parameters **detector_number** (*int*) – The detector number. Detectors numbers are one based

Returns Named tuple (DetConfMap) containing the detector configuration

Return type DetConfMap

get_detector_levels(*detector_number*)
Read detector levels

Parameters **detector_number** (*int*) – The number of the detector to get the levels of

Returns

DetLev named tuple containing the detector number, detector level, a list of status messages and a boolean that describes whether the detector is inhibited

Return type namedtuple

get_multiple_detector_levels(*detector_numbers*)
Get the levels for multiple detectors in one communication call

Parameters **detector_numbers** (*sequence*) – Sequence of integer detector numbers (remembe they are 1 based)

> **Warning:** This method uses "hidden" functions in the minimal modbus module for value conversion. As they are hidden, they are not guarantied to preserve their interface, which means that this method may break at any time

PyExpLabSys.drivers.crowcon.**main**()
Main function, used to simple functional test

7.9 The dataq_binary module

7.9.1 Autogenerated API documentation for dataq_binary

DataQ Binary protocol driver

To get started using the one of the supported DataQ data cards, use one of the sub-classes of DataQBinary e.g. DI1110, set scan list and start:

```
# Instantiate dataq object
dataq = DI1110('/dev/ttyUSB0')

# Get all available information from the data cards
print(repr(dataq.info()))

# Set sample rate frequence
dataq.sample_rate(1000)

# Set scan list (meaure on channel 1 first, then 0 and finnaly 2)
dataq.scan_list([1, 0, 2])

dataq.clear_buffer()
dataq.start()
sleep(0.1)
from pprint import pprint
try:
    while True:
        pprint(dataq.read())
        sleep(0.5)
except KeyboardInterrupt:
    dataq.stop()
else:
    dataq.stop()
dataq.clear_buffer()
```

If the data card is being used on the limit of emptying the data buffer before it overflow, it might be useful to put the calls to `read` in a try except:

```
while True:
    try:
        dataq.read()
    except dataq_binary.BufferOverflow:
        # Re-start i.e. stop and start
```

or, to simple start and stop the card for a short amount of time, if slow measurements, which would otherwise fill buffer are required:

```
while True:
    dataq.start()
    sleep(0.1)
    dataq.read()
    dataq.stop()
    # We really only need to measure every 10s
    sleep(10)
```

On some Linux system, at the end of 2017, some models, e.g. the DI-1110 wasn't automatically mounted. In that case, it can be manually mounted with a command like this one:

```
sudo modprobe usbserial vendor=0x0683 product=0x1110
```

On should be possible, on Debian based Linux systems to add an automatic mount rule along the lines of this thread: https://askubuntu.com/questions/525016/cant-open-port-dev-ttyusb0

So, add a new udev rule /etc/udev/rules.d/99-dataq_di1110.rules

with this content:

```
# /etc/udev/rules.d/99-dataq_di1110.rules
# contains DataQ DI-1110 udev rule to patch default
# rules
SYSFS{idProduct}=="1110",
SYSFS{idVendor}=="0683",
RUN+="/sbin/modprobe -q usbserial product=0x1110 vendor=0x0683"
```

and aftwerwards run: sudo udevadm control –reload-rules

exception PyExpLabSys.drivers.dataq_binary.**BufferOverflow**

Bases: exceptions.Exception

Custom exception to indicate a buffer overflow

class PyExpLabSys.drivers.dataq_binary.**DataQBinary**(*device*, *serial_kwargs=None*)

Bases: object

Base class for DataQBinary driver

end_char = '\r'

Serial communication end character

read_wait_time = 0.001

Wait time between serial write and read

infos = {'device name': 1, 'firmware revision': 2, 'sample rate divisor': 9, 'seria

Information items that can be retrieved along with their code number

led_colors = {'black': 0, 'blue': 1, 'cyan': 3, 'green': 2, 'magenta': 5, 'red':

Supported LED colors along with the code number

buffer_overflow_size = 4095

Buffer overflow size, may be overwritte in sub classes

packet_sizes = {16: 0, 32: 1, 64: 2, 128: 3, 256: 4, 512: 5, 1024: 6, 2048: 7}

Packet sizes and code numbers

__init__(*device*, *serial_kwargs=None*)

Initialize local variables

Parameters

- **device** (*str*) – The device e.g: '/dev/ttyUSB0' of 'COM1'
- **serial_kwargs** (*dict*) – dict of keyword arguments for serial.Serial

clear_buffer()
Clear the buffer

info(*info_name='all'*)
Return information about the device

Parameters **info_name** (*str*) – Name of the requested information item(s). If info_name is one the specific info names, (the keys in DataQBinary.infos), a string will be returned with the value. If info_name is 'all', all values will be returned in a dict

Returns Information items

Return type str or dict

start()
Start data acquisition

stop()
Stop data acquisition

This also implies clearing the buffer of any remaining data

scan_list(*scan_list*)
Set the scan list

The scan list is the list of inputs to acquire from on the data card. The scan list can hold up to 11 items, since there are a total on 11 inputs and each element can only be there once. The analogue input channel are numbered 0-7, 8 is the counter channel, 9 is the rate channel and 10 is the general purpose input channel. 0.7 are specified only by their number, 8, 9 and 10 are configured specially, which is not described here yet.

Parameters **scan_list** (*list*) – Etc. [3, 5, 0] for analogue input chanel 3, 5 and 0. NOTE: The numbers are integers, not strings.

sample_rate(*rate*)
Set the sample rate

The value values are calculated as being in the range of

sample rate divisor / 375 to sample rate divisor / 65535

So e.g. for the DI-1110 product, with a sample rate divisor of 60,000,000, the valid inputs are in range from 915.5413 to 160000.

Parameters **rate** (*float*) – The sample rate given in number of elements in the scan list sampled per second (i.e. in Hz). Valid values depend on the model and is given by the "sample rate divisor" information item (see the info method). See information about how to calculate the valid input range above.

packet_size(*size*)
Set the packet size

The packet size is he amount of data acquired before it is placed in the read buffer. The available packet sizes are the keys in packet_sizes class variable.

Parameters **size** (*int*) – The requested packet size

led_color(*color*)
Set the LED color

Parameters **color** (*str*) – The available colors are the keys in the led_colors class variable

read()
Read all values waiting

This method reads all available damples from the data card and returns for every channel the mean of those samples.

The returned data is on the form of a list with one item for each item in the scan list and in the same order. Each of the items is in them selves a dict, which holds the mean value of the samples, the number of samples in the mean, the channel number (0-based) and information about how full the data buffer was, at the time when it was read out. An example could be:

```
[{'buffer_status': '3040/4095 bytes',
  'channel': 1,
  'samples': 506,
  'value': -1.6224543756175889},
 {'buffer_status': '3040/4095 bytes',
  'channel': 0,
  'samples': 507,
  'value': -0.0044494267751479287},
 {'buffer_status': '3040/4095 bytes',
  'channel': 2,
  'samples': 507,
  'value': 1.6192735299556213}]
```

where the scan list was set to [1, 0, 2].

Returns

A list of values for each of the items in the scan-list and in the same order. For details of returns values see above.

Return type list

Raises ***BufferOverflow*** – If the buffer was full at the time of reading. The behavior in this case is ill-defined, so it is better to re-start the measurement if that happens.

class PyExpLabSys.drivers.dataq_binary.**DI1110**(*device*, *serial_kwargs=None*)
Bases: *PyExpLabSys.drivers.dataq_binary.DataQBinary*

FIXME

PyExpLabSys.drivers.dataq_binary.**module_test**()
Run primitive module tests

7.10 The dataq_comm module

7.10.1 Autogenerated API documentation for dataq_comm

Driver for DATAQ dac units

class PyExpLabSys.drivers.dataq_comm.**DataQ**(*port*)
Bases: object

driver for the DataQ Instrument

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
comm function

dataq()
Returns the string DATAQ

device_name()
Returns device name

firmware()
Returns firmware version

serial_number()
Returns device serial number

start_measurement()
Start a measurement scan

read_measurements()
Read the newest measurents

stop_measurement()
Stop a measurement scan

add_channel(*channel*)
Adds a channel to scan slist. So far only analog channels are accepted

set_ascii_mode()
change response mode to ACSII

set_float_mode()
change response mode to float

reset_scan_list()
Reseting the scan list

7.11 The deltaco_TB_298 module

7.11.1 Autogenerated API documentation for deltaco_TB_298

7.12 The edwards_agc module

7.12.1 Autogenerated API documentation for edwards_agc

Driver and simple test case for Edwards Active Gauge Controler

class PyExpLabSys.drivers.edwards_agc.**EdwardsAGC**(*port='/dev/ttyUSB0'*)
Bases: object

Primitive driver for Edwards Active Gauge Controler Complete manual found at http://www.idealvac.com/files/brochures/Edwards_AGC_D386-52-880_IssueM.pdf

__init__(*port='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Implements basic communication

gauge_type(*gauge_number*)
Return the type of gauge

read_pressure(*gauge_number*)
Read the pressure of a gauge

pressure_unit(*gauge_number*)
Read the unit of a gauge

current_error()
Read the current error code

software_version()
Return the software version of the controller

7.13 The edwards_nxds module

7.13.1 Autogenerated API documentation for edwards_nxds

Driver for Edwards, nXDS pumps

class PyExpLabSys.drivers.edwards_nxds.**EdwardsNxds**(*port*)
Bases: object

Driver for the Edwards nXDS series of dry pumps

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Ensures correct protocol for instrument

read_pump_type()
Read identification information

read_pump_temperature()
Read Pump Temperature

read_serial_numbers()
Read Pump Serial numbers

read_run_hours()
Return number of run hours

set_run_state(*on_state*)
Start or stop the pump

status_to_bin(*word*)
Convert status word to array of binaries

bearing_service()
Status of bearings

pump_controller_status()
Read the status of the pump controller

read_normal_speed_threshold()
Read the value for acknowledge the pump as normally running

read_standby_speed()
Read the procentage of full speed on standby

read_pump_status()
Read the overall status of the pump

read_service_status()
Read the overall status of the pump

set_standby_mode(*standbymode*)
Set the pump on or off standby mode

7.14 The epimax module

7.14.1 Autogenerated API documentation for epimax

Driver for the Epimax PVCi process vacuum controller

There are three controllers share the same kind of communication:

- PVCX
- PVCi
- PVCiDuo

The structure of the communication to these devices is the same and a part of the parameters are also the same, but there are also some parameters that differ. Therefore, the driver is implemented in such a way, that there is a base class (PVCCommon) that contains the communication functionality and the parameter from the common parameter definition. There can then be one class for each of the 3 specific devices, that adds in the parameters that are specific to this device. To see how that works, look at the *PVCi* class.

The implementation in this file is based on the documents:

- "EMComm MODBUS Communications Handbook" version 3.10
- "PVCX, PVCi & PVCiDuo EMComm Parameter List Handbook" version 3.00 (hereafter referred to as the parameter list)

Unfortunately, these documents are not (that I could find) available on the web and must be fetched by emailing Epimax support.

Note: At present only the PVCi driver is implemented and only partially

Note: At present no writing is implemented

class PyExpLabSys.drivers.epimax.**PVCCommon**(*port*, *slave_address=1*, *check_hardware_version=True*)
Bases: minimalmodbus.Instrument

Common base for the PVCX, PVCi and PVCiDuo devices

This common class must be sub-classed and the global_id and firmware_name class variables overwritten and the self.fields dict updated if necessary. See the *PVCi* inplementation for details.

All requests for values (parameters) goes via value field names. To get a list of the available fields, have a look at the keys in the fields dict of the common class and the sub-class. These fields names can then be used with *get_field()* and *get_fields()* method or accessed as if they were attributes of the class.

Remember to call .close() after use.

__init__(*port*, *slave_address=1*, *check_hardware_version=True*)
Initialize communication

Parameters

- **port** (*unicode*) – The port specification of the device e.g. '/dev/????'
- **slave_address** (*int*) – The address of the slave device, default is 1
- **check_hardware_version** (*bool*) – Indicated whether a check should be performed for correct hardware at __init__ time

close()
Close the serial connection

get_field(*field_name*)
Return the value for the field named field_name

Parameters **field_name** (*str*) – The name of the field to get. The names used are adapted parameter names from the command list turned. See the keys in `fields` to see all possible values.

Returns An object with type corresponding to the value (int, float or str)

Return type object

Raises **KeyError** – If the requested field_name is unknown

get_fields(*fields='common'*)
Return a dict with fields and values for a list of fields

This method is specifically for getting multiple values in the shortest amount of time. It works by always reading the maximum amount of registers (32) at a time and then using the remaining payload for subsequent values if they happen to be contained in the registers that have already been read.

Parameters **fields** (*sequence or unicode*) – A sequence (list, tuple) of fields names or 'common' which indicates fields with an address between 0x80 and 0x9E (this is the default) or 'all'.

Returns Field name to value mapping

Return type dict

class PyExpLabSys.drivers.epimax.**PVCi**(**args*, ***kwargs*)
Bases: *PyExpLabSys.drivers.epimax.PVCCommon*

Driver for the PVCi device

For details of the functionality of this driver, see the docstring for the common base class *PVCCommon*

__init__(**args*, ***kwargs*)
For specification for __init__ arguments, see *PVCCommon.__init__()*

PyExpLabSys.drivers.epimax.**bytes_to_firmware_version**(*bytes_*)
Convert 4 bytes to firmware type and version

PyExpLabSys.drivers.epimax.**bytes_to_string**(*bytes_*, *valid_chars=None*)
Convert the 16 bit integer values from registers to a string

Parameters **valid_chars** (*sequence*) – Sequence of two integers indicating the start and end of a range of valid bytes (both values included). All chars outside the range will be filtered out.

PyExpLabSys.drivers.epimax.**bytes_to_float**(*bytes_*)
Convert 2 16 bit registers to a float

PyExpLabSys.drivers.epimax.**bytes_to_slot_id**(*bytes_*)
Convert 4 bytes to the slot ID

PyExpLabSys.drivers.epimax.**bytes_to_status**(*bytes_*, *status_type*)
Convert bytes to trip and digital input statuses

PyExpLabSys.drivers.epimax.**byte_to_bits**(*byte*)
Convert a byte to a list of bits

PyExpLabSys.drivers.epimax.**raise_if_not_set**(*bits*, *index*, *parameter*)
Raise a ValueError if bit is not set

PyExpLabSys.drivers.epimax.**ion_gauge_status**(*bytes_*, *controller_type=None*)
Read of ion gauge status

PyExpLabSys.drivers.epimax.**bytes_to_bakeout_flags**(*bytes_*)
Returns the bakeout flags from bytes

PyExpLabSys.drivers.epimax.**run_module**()
Tests basic functionality

Will init a PVCi on USB0 and out all info fields and gauge 1 pressure and bakeout info continuously

7.15 The four_d_systems_1 module

7.15.1 Autogenerated API documentation for four_d_systems_1

7.16 The four_d_systems_2 module

7.16.1 Autogenerated API documentation for four_d_systems_2

7.17 The freescale_mma7660fc module

7.17.1 Autogenerated API documentation for freescale_mma7660fc

Driver for AIS328DQTR 3 axis accelerometer

class PyExpLabSys.drivers.freescale_mma7660fc.**MMA7660FC**
Bases: object

Class for reading accelerometer output

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

read_values()
Read a value from the sensor

7.18 The fug module

7.18.1 Autogenerated API documentation for fug

Driver for "fug NTN 140 - 6,5 17965-01-01" power supply Communication via the Probus V serial interface.

Written using the two documents:

1. Interface system Probus V - Documentation for RS232/RS422 Revision of document 2.4

2. Probus V - Command Reference Base Module ADDAT30 Firmware PIC0162 V4.0 Version of Document V2.22

Should be freely available from http://www.fug-elektronik.de/en/support/download.html (Available August 25 2017)

class PyExpLabSys.drivers.fug.**FUGNTN140Driver**(*port='/dev/ttyUSB0'*, *baudrate=9600*, *parity='N'*, *stopbits=1*, *bytesize=8*, *device_reset=True*, *V_max=6.5*, *I_max=10*)

Bases: object

Driver for fug NTN 140 power supply

Methods

- **Private**
  - __init__
  - _check_answer
  - _flush_answer
  - _get_answer
  - _write_register
  - _read_register
- **Public**
  - reset()
  - stop()
  - is_on()
  - output(state=True/False)
  - get_state()
  - identification_string()
  - —
  - set_voltage(value)
  - get_voltage()
  - monitor_voltage()
  - ramp_voltage(value, program=0)
  - ramp_voltage_running()
  - —
  - set_current(value)
  - get_current()
  - monitor_current()
  - ramp_current(value, program=0)
  - ramp_current_running()

__init__(*port='/dev/ttyUSB0'*, *baudrate=9600*, *parity='N'*, *stopbits=1*, *bytesize=8*, *device_reset=True*, *V_max=6.5*, *I_max=10*)

Initialize object variables

For settings port, baudrate, parity, stopbits, bytesize, see the pyserial documentation.

Parameters **device_reset** (*bool*) – If true, resets all device parameters to default values

reset()
Resets device

stop(*reset=True*)
Closes device properly before exit

is_on()
Checks if output is ON (>DON) Returns True if ON

output(*state=False*)
Set output ON (>BON)

get_state()
Checks whether unit is in CV or CC mode (>DVR/>DIR)

identification_string()
Output serial number of device

set_voltage(*value*)
Sets the voltage (>S0)

get_voltage()
Reads the set point voltage (>S0A)

monitor_voltage()
Reads the actual (monitor) voltage (>M0)

ramp_voltage(*value*, *program=0*)
Activates ramp function for voltage value : ramp value in volts/second (>S0R)

program	setvalue behaviour
0	(default) no ramp function. Setpoint is implemented immediately
1	>S0A follows the value in >S0 with the adjusted ramp rate upwards and downwards
2	>S0A follows the value in >S0 with the adjusted ramp rate only upwards. When programming downwards, >S0A follows >S0 immediately.
3	>S0A follows the value in >S0 with a special ramp function only upwards. When programming downwards, >S0A follows >S0 immediately. Ramp between 0..1 with 11.11E-3 per second. Above 1 : with >S0R
4	Same as 2, but >S0 as well as >S0A are set to zero if >DON is 0

ramp_voltage_running()
Return status of voltage ramp. True: still ramping False: ramp complete

set_current(*value*)
Sets the current (>S1)

get_current()
Reads the set point current (>S1A)

monitor_current()
Reads the actual (monitor) current (>M1)

ramp_current(*value*, *program=0*)
Activates ramp function for current. See ramp_voltage() for description.

ramp_current_running()
Return status of current ramp. True: still ramping False: ramp complete

read_H1()
Read H1 FIXME not yet done

print_states(*t0=0*)
Print the current state of the power supply

PyExpLabSys.drivers.fug.**test**()
Module test function

7.19 The galaxy_3500 module

7.19.1 Autogenerated API documentation for galaxy_3500

Python interface for Galaxy 3500 UPS. The driver uses the telnet interface of the device.

class PyExpLabSys.drivers.galaxy_3500.**Galaxy3500**(*hostname*)
Bases: object

Interface driver for a Galaxy3500 UPS.

__init__(*hostname*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*, *keywords=None*)
Send a command to the ups

alarms()
Return list of active alarms

battery_charge()
Return the battery charge state

temperature()
Return the temperature of the UPS

battery_status()
Return the battery voltage

output_measurements()
Return status of the device's output

input_measurements()
Return status of the device's output

7.20 The honeywell_6000 module

7.20.1 Autogenerated API documentation for honeywell_6000

Driver for HIH6000 class temperature and humidity sensors

class PyExpLabSys.drivers.honeywell_6000.**HIH6130**
Bases: object

Class for reading pressure and temperature from Honeywell HumidIcon HIH-6130/6131

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

read_values()
Read a value from the sensor

7.21 The inficon_sqm160 module

7.21.1 Autogenerated API documentation for inficon_sqm160

Driver for Inficon SQM160 QCM controller

class PyExpLabSys.drivers.inficon_sqm160.**InficonSQM160**(*port='/dev/ttyUSB0'*)
Bases: object

Driver for Inficon SQM160 QCM controller

__init__(*port='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Implements actual communication with device

static crc_calc(*input_string*)
Calculate crc value of command

show_version()
Read the firmware version

show_film_parameters()
Read the film paramters

rate(*channel=1*)
Return the deposition rate

thickness(*channel=1*)
Return the film thickness

frequency(*channel=1*)
Return the frequency of the crystal

crystal_life(*channel=1*)
Read crystal life

7.22 The innova module

7.22.1 Autogenerated API documentation for innova

Driver for the Innova RT 6K UPS

Implemented from this document: http://networkupstools.org/protocols/megatec.htmlL

PyExpLabSys.drivers.innova.**STATUS_INQUIRY_NAMES = ['input_voltage', 'input_fault_voltage',**
The first 7 places of the response to the status inquiry are numbers, who are paired with the names in the list below

PyExpLabSys.drivers.innova.**STATUS_INQUIRY_BOOLEANS = ['utility_fail_immediate', 'battery_lo**
The last section of the response to the status inquiry are 0's and 1's, which indicate the boolean status of the fields listed below.

PyExpLabSys.drivers.innova.**RATING_INFORMATION_FIELDS = ['rating_voltage', 'rating_current',**
The names for the floats returned as section from the rating information command

class PyExpLabSys.drivers.innova.**Megatec**(*device*, *baudrate=2400*, *timeout=2.0*)
Bases: object

Driver that implements parts of the Megatech specification

__init__(*device*, *baudrate=2400*, *timeout=2.0*)
x.__init__(...) initializes x; see help(type(x)) for signature

com(*command*)
Perform communication

get_status()
Return the status as a dict

The values in the dict are either float or booleans. The keys for the float values are:

- output_voltage
- input_voltage
- temperature_C
- input_frequency
- battery_voltage
- output_current_load_percent
- input_fault_voltage

The keys for the boolean values are:

- utility_fail_immediate
- battery_low
- bypass_boost_or_buck_active
- UPS_failed
- UPS_type_is_standby
- test_in_progress
- shutdown_active
- beeper_on

test_for_10_sec()
Run a test of the batteries for 10 sec and return to utility

ups_information()
Return the UPS information

ups_rating_information()
Return the UPS rating information as a dict

The dict contains float valus for the following 4 fields:

- battery_voltage

- frequency
- rating_current
- rating_voltage

class PyExpLabSys.drivers.innova.**InnovaRT6K**(*device*, *baudrate=2400*, *timeout=2.0*)
Bases: *PyExpLabSys.drivers.innova.Megatec*

for the InnovaRT6k UPS

7.23 The intellemetrics_il800 module

7.23.1 Autogenerated API documentation for intellemetrics_il800

Driver for IL800 deposition controller

class PyExpLabSys.drivers.intellemetrics_il800.**IL800**(*port*)
Bases: object

Driver for IL800 depostition controller

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Communicate with instrument

rate()
Return the deposition rate in nm/s

thickness()
Return the currently measured thickness in nm

frequency()
Return the qrystal frequency in Hz

7.24 The isotech_ips module

7.24.1 Autogenerated API documentation for isotech_ips

Driver for ISO-TECH IPS power supply series

It has not been possible to get the device to give any meaningfull replys, but actually setting output values works.

class PyExpLabSys.drivers.isotech_ips.**IPS**(*port*)
Bases: object

Driver for IPS power supply

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Communicate with instrument

set_vlimit_to_max()
Set the voltage limit to the maximum the device deliver

set_ilimit_to_max()
Set the current limit to the maximum the device deliver

set_relay_status(*status=False*)
Turn the output on or off

set_output_voltage(*voltage*)
Set the output voltage

set_voltage_limit(*voltage*)
Set the voltage limit

set_current_limit(*current*)
Set the current limit

7.25 The keithley_2700 module

7.25.1 Autogenerated API documentation for keithley_2700

Simple driver for Keithley Model 2700

class PyExpLabSys.drivers.keithley_2700.**KeithleySMU**(*interface*, *device='/dev/ttyUSB0'*)

Bases: *PyExpLabSys.drivers.scpi.SCPI*

Simple driver for Keithley Model 2700

__init__(*interface*, *device='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

select_measurement_function(*function*)
Select a measurement function.

Keyword arguments: Function – A string stating the wanted measurement function.

read()
Read a value from the device

7.26 The keithley_smu module

7.26.1 Autogenerated API documentation for keithley_smu

Simple driver for Keithley SMU

class PyExpLabSys.drivers.keithley_smu.**KeithleySMU**(*interface*, *hostname=''*, *device=''*, *baudrate=9600*)

Bases: *PyExpLabSys.drivers.scpi.SCPI*

Simple driver for Keithley SMU

__init__(*interface*, *hostname=''*, *device=''*, *baudrate=9600*)
x.__init__(...) initializes x; see help(type(x)) for signature

output_state(*output_on=False*, *channel=1*)
Turn the output on or off

set_current_measure_range(*current_range=None*, *channel=1*)
Set the current measurement range

set_integration_time(*nplc=None*, *channel=1*)
Set the measurement integration time

read_current(*channel=1*)
Read the measured current

read_voltage(*channel=1*)
Read the measured voltage

set_current_limit(*current*, *channel=1*)
Set the desired current limit

set_voltage(*voltage*, *channel=1*)
Set the desired voltage

set_voltage_limit(*voltage*, *channel=1*)
Set the desired voltate limit

set_current(*current*, *channel=1*)
Set the desired current

iv_scan(*v_from*, *v_to*, *steps*, *settle_time*, *channel=1*)
Perform iv_scan

7.27 The kjlc_pressure_gauge module

7.27.1 Autogenerated API documentation for kjlc_pressure_gauge

Module contains driver for KJLC 3000 pressure gauge

class PyExpLabSys.drivers.kjlc_pressure_gauge.**KJLC300**(*port*)
Bases: object

Class implements a KJLC interface

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

close()
Closes connection

read_software_version()
Reads software version

read_pressure()
Reads pressure in Torr

7.28 The lascar module

7.28.1 Autogenerated API documentation for lascar

Driver for the EL-USB-RT temperature and humidity USB device from Lascar

Calling read on the device will return either the temperature or the humidity.

If the first byte is it is a temperature. The next 2 bytes is a unsigned integer which, is used to calculate the temperature as:

temp = -100 * 0.1 * (unsigned_short)

If the first byte is it is humidity. The next byte is an unsigned char, which is used to calculate the relative humidity as:

humidity = 0.5 * (unsigned_char)

class PyExpLabSys.drivers.lascar.**ElUsbRt**(*device_path=None*)
Bases: object

Driver for the EL-USB-RT device

__init__(*device_path=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

get_temperature_and_humidity()
Returns the temperature (in celcius, float) and relative humidity (in %, float) in a dict

get_temperature()
Returns the temperature (in celcius, float)

7.29 The microchip_tech_mcp3428 module

7.29.1 Autogenerated API documentation for microchip_tech_mcp3428

Driver for Microchip Technology MCP3428 Analog Input device Calibrated to PR33-13 from ncd.io other products will use different voltage references.

class PyExpLabSys.drivers.microchip_tech_mcp3428.**I2C**(*device*, *bus*)
File based i2c. Code adapted from: https://www.raspberrypi.org/forums/viewtopic.php?t=134997

__init__(*device*, *bus*)

write(*values*)
Write a value to i2c port

read(*number_of_bytes*)
Read value from i2c port

close()
Close the device

class PyExpLabSys.drivers.microchip_tech_mcp3428.**MCP3428**
Bases: object

Class for reading voltage from MCP3428 For some reason this chip works only partly with smbus, hence the use of file based i2c.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

read_sample(*channel=1*, *gain=1*, *resolution=12*)
Read a single sample

gain(*gain=1*)
Return the command code to set gain

resolution(*resolution=12*)
Return the command code to set resolution

channel(*channel=1*)
Return the command code to set channel

7.30 The mks_925_pirani module

7.30.1 Autogenerated API documentation for mks_925_pirani

Driver for MKS 925 micro pirani

class PyExpLabSys.drivers.mks_925_pirani.**Mks925**(*port*)
Bases: object

Driver for MKS 925 micro pirani

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Implement communication protocol

read_pressure()
Read the pressure from the device

set_comm_speed(*speed*)
Change the baud rate

change_unit(*unit*)
Change the unit of the return value

read_serial()
Read the serial number of the device

7.31 The mks_937b module

7.31.1 Autogenerated API documentation for mks_937b

Driver for MKS 937b gauge controller

class PyExpLabSys.drivers.mks_937b.**Mks937b**(*port*)
Bases: object

Driver for MKS 937B Gauge Controller

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Implement communication protocol

read_pressure_gauge(*gauge_number*)
Read a specific pressure gauge

read_sensor_types()
Return a list of connected sensors

read_all_pressures()
Returns an overview of all sensors

pressure_unit(*unit=None*)
Read or configure pressure unit Legal values: torr, mbar, pascal, micron

7.32 The mks_g_series module

7.32.1 Autogenerated API documentation for mks_g_series

Driver for MKS g-series flow controller

class PyExpLabSys.drivers.mks_g_series.**MksGSeries**(*port='/dev/ttyUSB0'*)
Driver for G-series flow controllers from MKS

__init__(*port='/dev/ttyUSB0'*)

checksum(*command*)
Calculate checksum of command

comm(*command*, *addr*)
Implements communication protocol

read_full_scale_range(*addr*)
Read back the current full scale range from the instrument

read_device_address(*address=254*)
Read the device address

set_device_address(*old_addr*, *new_addr*)
Set the device address

read_current_gas_type(*addr*)
Read the current default gas type

read_run_hours(*addr*)
Return number of running hours of mfc

read_setpoint(*addr*)
Read current setpoint

set_flow(*value*, *addr=254*)
Set the flow setpoint

purge(*t=1*, *addr=254*)
purge for t seconds, default is 1 second

read_flow(*addr=254*)
Read the flow

read_serial_number(*addr=254*)
Read the serial number of the device

7.33 The mks_pi_pc module

7.33.1 Autogenerated API documentation for mks_pi_pc

7.34 The omega_D6400 module

7.34.1 Autogenerated API documentation for omega_D6400

Driver for Omega D6400 daq card

class PyExpLabSys.drivers.omega_D6400.**OmegaD6400**(*address=1*, *port='/dev/ttyUSB0'*)
Bases: object

Driver for Omega D6400 daq card

__init__(*address=1*, *port='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*, *value=None*)
Communicates with the device

read_value(*channel*)
Read a measurement value from a channel

read_address()
Read the RS485 address of the device

write_enable()
Enable changes to setup values

range_codes(*fullrange=0*, *action=None*)
Returns the code corresponding to a given range

update_range_and_function(*channel*, *fullrange=None*, *action=None*)
Set the range and measurement type for a channel

7.35 The omega_cn7800 module

7.35.1 Autogenerated API documentation for omega_cn7800

Omega CN7800 Modbus driver. Might also work with other CN units

class PyExpLabSys.drivers.omega_cn7800.**CN7800**(*port*)
Bases: object

Driver for the omega CN7800

__init__(*port*)
x.__init__(...) initializes x; see help(type(x)) for signature

read_temperature()
Read the temperature from the device

7.36 The omega_cni module

7.36.1 Autogenerated API documentation for omega_cni

This module contains drivers for equipment from Omega. Specifically it contains a driver for the ??? thermo couple read out unit.

class PyExpLabSys.drivers.omega_cni.**ISeries**(*port*, *baudrate=19200*, *comm_stnd='rs232'*)
Bases: object

Driver for the iSeries omega temperature controllers

__init__(*port*, *baudrate=19200*, *comm_stnd='rs232'*)
Initialize internal parameters

Parameters **port** – A serial port designation as understood by pySerial

command(*command*, *response_length=None*, *address=None*)
Run a command and return the result

Parameters

- **command** (*str*) – The command to execute
- **response_length** (*int*) – The expected legth of the response. Will force the driver to wait untill this many characters is ready as a response from the device.

reset_device(*address=None*)
Reset the device

identify_device(*address=None*)
Return the identity of the device

read_temperature(*address=None*)
Return the temperature

close()
Close the connection to the device

class PyExpLabSys.drivers.omega_cni.**CNi3244_C24**(*port*)
Bases: *PyExpLabSys.drivers.omega_cni.ISeries*

Driver for the CNi3244_C24 device

__init__(*port*)
Initialize internal parameters

Parameters **port** – A serial port designation as understood by pySerial

7.37 The omegabus module

7.37.1 Autogenerated API documentation for omegabus

Driver for OmegaBus devices

class PyExpLabSys.drivers.omegabus.**OmegaBus**(*device='/dev/ttyUSB0'*, *model='D5251'*, *baud=300*)
Bases: object

Driver for OmegaBus devices

__init__(*device='/dev/ttyUSB0'*, *model='D5251'*, *baud=300*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Handles serial protocol

read_value(*channel*, *convert_to_celcius=True*)
Read the measurement value

read_max(*channel*)
The maximum read-out value

read_min(*channel*)
The minimum read-out value

read_setup()
Read Device setup information

7.38 The omron_d6fph module

7.38.1 Autogenerated API documentation for omron_d6fph

Hint for implementation found at http://forum.arduino.cc/index.php?topic=285116.0

class PyExpLabSys.drivers.omron_d6fph.**OmronD6fph**
Bases: object

Class for reading pressure and temperature from Omron D6F-PH series Ranging not implemented for all models

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

init_device()
Sensor needs to be initiated after power up

read_value(*command*)
Read a value from the sensor

read_pressure()
Read the pressure value

read_temperature()
Read the temperature value

7.39 The pfeiffer_qmg420 module

7.39.1 Autogenerated API documentation for pfeiffer_qmg420

7.40 The pfeiffer_qmg422 module

7.40.1 Autogenerated API documentation for pfeiffer_qmg422

This module contains the driver code for the QMG422 control box for a pfeiffer mass-spectrometer. The code should in principle work for multiple type of electronics. It has so far been tested with a qme-125 box and a qme-??? box. The module is ment as a driver and has very little function in itself. The module is ment to be used as a sub-module for a large program providing the functionality to actually use the mass-spectrometer.

Known bugs: Not all code has a proper distinction between the various electronics. The qme-125 has many limitations compared to the qme-??? and these limitations are not always properly expressed in the code or the output of the module

class PyExpLabSys.drivers.pfeiffer_qmg422.**qmg_422**(*port='/dev/ttyS0'*, *speed=19200*)
Bases: object

The actual driver class.

__init__(*port='/dev/ttyS0'*, *speed=19200*)
Initialize the module

comm(*command*)
Communicates with Baltzers/Pferiffer Mass Spectrometer

Implements the low-level protocol for RS-232 communication with the instrument. High-level protocol can be implemented using this as a helper

Parameters **command** (*str*) – The command to send

Returns The reply associated with the last command

Return type str

communication_mode(*computer_control=False*)
Returns and sets the communication mode.

Parameters **computer_control** (*bool*) – Activates ASCII communication with the device

Returns The current communication mode

Return type str

simulation()
Chekcs wheter the instruments returns real or simulated data

Returns Message telling whether the device is in simulation mode

Return type str

set_channel(*channel*)
Set the current channel :param channel: The channel number :type channel: integer

read_sem_voltage()
Read the SEM Voltage :return: The SEM voltage :rtype: str

read_preamp_range()
Read the preamp range This function is not fully implemented :return: The preamp range :rtype: str

read_timestep()
Reads the integration period of a measurement :return: The integration period in non-physical unit :rtype: str

sem_status(*voltage=-1*, *turn_off=False*, *turn_on=False*)
Get or set the SEM status :param voltage: The wanted SEM-voltage :type voltage: integer :param turn_off: If True SEM will be turned on (unless turn_of==True) :type turn_off: boolean :param turn_on: If True SEM will be turned off (unless turn_on==True) :type turn_on: boolean :return: The SEM voltage, The SEM status, True means voltage on :rtype: integer, boolan

emission_status(*current=-1*, *turn_off=False*, *turn_on=False*)
Get or set the emission status. :param current: The wanted emission status. Only works for QME??? :type current: integer :param turn_off: If True emission will be turned on (unless turn_of==True) :type turn_off: boolean :param turn_on: If True emission will be turned off (unless turn_on==True) :type turn_on: boolean :return: The emission value (for QME???), The emission status, True means filament on :rtype: integer, boolan

detector_status(*SEM=False*, *faraday_cup=False*)
Choose between SEM and Faraday cup measurements

read_voltages()
Read the qme-voltages

update_state()
Update the knowledge of the internal knowledge of the instrument

start_measurement()
Start the measurement

actual_range(*amp_range*)
Returns the range that should be send to achieve the desired range

get_single_sample()
Read a single sample from the device

get_multiple_samples(*number*)
Read multiple samples from the device

config_channel(*channel*, *mass=-1*, *speed=-1*, *enable=''*, *amp_range=0*)
Config a MS channel for measurement

measurement_running()
Check if a measurement is running

waiting_samples()
Return number of waiting samples

mass_scan(*first_mass*, *scan_width*, *amp_range=0*)
Setup the mass spec for a mass scan

mass_time(*ns*)
Setup the mass spec for a mass-time measurement

7.41 The pfeiffer_turbo_pump module

7.41.1 Autogenerated API documentation for pfeiffer_turbo_pump

Self contained module to run a Pfeiffer turbo pump including fall-back text gui and data logging.

class PyExpLabSys.drivers.pfeiffer_turbo_pump.**CursesTui**(*turbo_instance*)
Bases: threading.Thread

Text gui for controlling the pump

__init__(*turbo_instance*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Cleanup terminal

class PyExpLabSys.drivers.pfeiffer_turbo_pump.**TurboReader**(*turbo_instance*)
Bases: threading.Thread

Keeps track of all data from a turbo pump with the intend of logging them

__init__(*turbo_instance*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class PyExpLabSys.drivers.pfeiffer_turbo_pump.**TurboLogger**(*turboreader*, *parameter*, *maximumtime=600*)
Bases: threading.Thread

Read a specific value and determine whether it should be logged

__init__(*turboreader*, *parameter*, *maximumtime=600*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

read_value()
Read the value of the logger

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class PyExpLabSys.drivers.pfeiffer_turbo_pump.**TurboDriver**(*adress=1*, *port='/dev/ttyUSB3'*)

Bases: threading.Thread

The actual driver that will communicate with the pump

__init__(*adress=1*, *port='/dev/ttyUSB3'*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

comm(*command*, *read=True*)
Implementaion of the communication protocol with the pump. The function deals with common syntax need for all commands.

Parameters

- **command** (*str*) – The command to send to the pump
- **read** (*Boolean*) – If True, read only not action performed

Returns The reply from the pump

Return type Str

crc_calc(*command*)
Helper function to calculate crc for commands :param command: The command for which to calculate crc :type command: str :return: The crc value :rtype: Str

read_rotation_speed()
Read the rotational speed of the pump

Returns The rotaional speed in Hz

Return type Float

read_set_rotation_speed()
Read the intended rotational speed of the pump

Returns The intended rotaional speed in Hz

Return type Int

read_operating_hours()
Read the number of operating hours

Returns Number of operating hours

Return type Int

read_gas_mode()
Read the gas mode :return: The gas mode :rtype: Str

read_vent_mode()
Read the venting mode :return: The venting mode :rtype: Str

read_sealing_gas()
Read whether sealing gas is applied :return: The sealing gas mode :rtype: Str

is_pump_accelerating()
Read if pump is accelerating :return: True if pump is accelerating, false if not :rtype: Boolean

turn_pump_on(*off=False*)
Spin the pump up or down :param off: If True the pump will spin down :type off: Boolean :return: Always returns True :rtype: Boolean

read_temperature()
Read the various measured temperatures of the pump :return: Dictionary with temperatures :rtype: Dict

read_drive_power()
Read the current power consumption of the pump :return: Dictionary containing voltage, current and power :rtype: Dict

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

7.42 The polyscience_4100 module

7.42.1 Autogenerated API documentation for polyscience_4100

Driver and test case for Polyscience 4100

class PyExpLabSys.drivers.polyscience_4100.**Polyscience4100**(*port='/dev/ttyUSB0'*)
Bases: object

Driver for Polyscience 4100 chiller

__init__(*port='/dev/ttyUSB0'*)
x.__init__(...) initializes x; see help(type(x)) for signature

comm(*command*)
Send serial commands to the instrument

set_setpoint(*value*)
Set the temperature setpoint

turn_unit_on(*turn_on*)
Turn on or off the unit

read_setpoint()
Read the current value of the setpoint

read_unit()
Read the measure unit

read_temperature()
Read the actual temperature of the water

read_pressure()
Read the output pressure

read_flow_rate()
Read the flow rate

read_ambient_temperature()
Read the ambient temperature in the device

read_status()
Answers if the device is turned on

7.43 The rosemount_nga2000 module

7.43.1 Autogenerated API documentation for rosemount_nga2000

7.44 The scpi module

7.44.1 Autogenerated API documentation for scpi

Implementation of SCPI standard

class PyExpLabSys.drivers.scpi.**SCPI**(*interface*, *device=''*, *tcp_port=5025*, *hostname=''*, *baudrate=9600*, *visa_string=''*, *line_ending='r'*)
Bases: object

Driver for scpi communication

__init__(*interface*, *device=''*, *tcp_port=5025*, *hostname=''*, *baudrate=9600*, *visa_string=''*, *line_ending='\r'*)
x.__init__(...) initializes x; see help(type(x)) for signature

scpi_comm(*command*, *expect_return=False*)
Implements actual communication with SCPI instrument

read_software_version()
Read version string from device

reset_device()
Rest device

device_clear()
Stop current operation

clear_error_queue()
Clear error queue

7.45 The specs_XRC1000 module

7.45.1 Autogenerated API documentation for specs_XRC1000

Self contained module to run a SPECS sputter gun including fall-back text gui

class PyExpLabSys.drivers.specs_XRC1000.**CursesTui**(*sourcecontrol*)
Bases: threading.Thread

Defines a fallback text-gui for the source control.

__init__(*sourcecontrol*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Cleanup the terminal

class PyExpLabSys.drivers.specs_XRC1000.**XRC1000**(*port=None*)
Bases: threading.Thread

Driver for X-ray Source Control - XRC 1000

__init__(*port=None*)
Initialize module

Establish serial connection and create status variable to expose the status for the instrument for the various gui's

comm(*command*)
Communication with the instrument

Implements the synatx need to send commands to instrument including handling carrige returns and extra lines of 'OK' and other pecularities of the instrument.

Parameters **command** (*str*) – The command to send

Returns The reply to the command striped for protocol technicalities

Return type str

posible comands: REM?, IEM?, UAN?, IHV?, IFI?, UFI?, PAN?, SERNO?, ANO?, STAT?, OPE? REM, LOC, IEM 20e-3, UAN 10e3, OFF, COOL, STAN, UAON, OPE, ANO 1, ANO 2

read_water_flow()
read the water flow from external hardware :return: water flow in L/min :rtype float

read_emission_current()
Read the emission current. Unit A :return: The emission current :rtype: float

read_filament_voltage()
Read the filament voltage. Unit V :return: The filament voltage :rtype: float

read_filament_current()
Read the filament current. Unit A :return: The filament current :rtype: float

read_anode_voltage()
Read the anode voltage. Unit V :return: The anode voltage :rtype: float

read_anode_power()
Read the anode voltage. Unit W :return: The anode voltage :rtype: float

standby()
Set the device on standby The function is not working entirely as intended. TODO: Implement check to see if the device is alrady in standby :return: The direct reply from the device :rtype: str

operate()
Set the device in operation mode TODO: This function should only be activated from standby!!! :return: The direct reply from the device :rtype: str

remote_enable(*local=False*)
Enable or disable remote mode :param local: If True the device is set to local, otherwise to remote :type local: Boolean :return: The direct reply from the device :rtype: str

change_control()
Enable or disable remote mode :param local: If True the device is set to local, otherwise to remote :type local: Boolean :return: The direct reply from the device :rtype: str

cooling()
Enable or disable water cooling :type local: Boolean :return: The direct reply from the device :rtype: str

update_status()
Update the status of the instrument

Runs a number of status queries and updates self.status

Returns The direct reply from the device

Return type str

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

7.46 The specs_iqe11 module

7.46.1 Autogenerated API documentation for specs_iqe11

Self contained module to run a SPECS sputter gun including fall-back text gui

class PyExpLabSys.drivers.specs_iqe11.**CursesTui**(*sputtergun*)
Bases: threading.Thread

Defines a fallback text-gui for the sputter gun.

__init__(*sputtergun*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()
Cleanup the terminal

class PyExpLabSys.drivers.specs_iqe11.**Puiqe11**(*simulate=False*)
Bases: threading.Thread

Driver for ion sputter guns from SPECS

__init__(*simulate=False*)
Initialize module

Establish serial connection and create status variable to expose the status for the instrument for the various gui's

comm(*command*)
Communication with the instrument

Implements the synatx need to send commands to instrument including handling carrige returns and extra lines of 'OK' and other pecularities of the instrument.

Parameters **command** (*str*) – The command to send

Returns The reply to the command striped for protocol technicalities

Return type str

read_sputter_current()
Read the sputter current. Unit mA :return: The sputter current :rtype: float

read_filament_voltage()
Read the filament voltage. Unit V :return: The filament voltage :rtype: float

read_filament_current()
Read the filament current. Unit A :return: The filament current :rtype: float

read_emission_current()
Read the emission current. Unit mA :return: The emission current :rtype: float

read_acceleration_voltage()
Read the acceleration voltage. Unit V :return: The acceleration voltage :rtype: float

read_temperature_energy_module()
Read the temperature of the electronics module This value is not extremely correct, use only as guideline. :return: The temperature :rtype: float

standby()
Set the device on standby The function is not working entirely as intended. TODO: Implement check to see if the device is alrady in standby :return: The direct reply from the device :rtype: str

operate()
Set the device in operation mode TODO: This function should only be activated from standby!!! :return: The direct reply from the device :rtype: str

remote_enable(*local=False*)
Enable or disable remote mode :param local: If True the device is set to local, otherwise to remote :type local: Boolean :return: The direct reply from the device :rtype: str

update_status()
Update the status of the instrument

Runs a number of status queries and updates self.status

Returns The direct reply from the device

Return type str

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

7.47 The srs_sr630 module

7.47.1 Autogenerated API documentation for srs_sr630

Driver for Standford Research Systems, Model SR630

class PyExpLabSys.drivers.srs_sr630.**SRS_SR630**(*port*)
Driver for Standford Research Systems, Model SR630

__init__(*port*)

comm(*command*)
Ensures correct protocol for instrument

config_analog_channel(*channel*, *follow_temperature=False*, *value=0*)
Configure an analog out channel

set_unit(*channel*, *unit*)
Set the measurement unit for a channel

tc_types()
List all configuration of all channels

read_open_status()
Check for open output on all channels

read_serial_number()
Return the serial number of the device

read_channel(*channel*)
Read the actual value of a channel

7.48 The stahl_hv_400 module

7.48.1 Autogenerated API documentation for stahl_hv_400

Driver for Stahl HV 400 Ion Optics Supply

class PyExpLabSys.drivers.stahl_hv_400.**StahlHV400**(*port='/dev/ttyUSB0'*)
Bases: object

Driver for Stahl HV 400 Ion Optics Supply

__init__(*port='/dev/ttyUSB0'*)
Driver for Stahl HV 400 Ion Optics Supply

comm(*command*)
Perform actual communication with instrument

identify_device()
Return the serial number of the device

query_voltage(*channel*)
Something is all wrong here...

set_voltage(*channel*, *value*)
Set the voltage of a channel

read_temperature()
Read temperature of device

check_channel_status()
Check status of channel

7.49 The stmicroelectronics_ais328dq module

7.49.1 Autogenerated API documentation for stmicroelectronics_ais328dq

Driver for STMicroelectronics AIS328DQTR 3 axis accelerometer

class PyExpLabSys.drivers.stmicroelectronics_ais328dq.**AIS328DQTR**
Bases: object

Class for reading accelerometer output

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

who_am_i()
Device identification

read_values()
Read a value from the sensor

7.50 The stmicroelectronics_l3g4200d module

7.50.1 Autogenerated API documentation for stmicroelectronics_l3g4200d

Driver for STMicroelectronics L3G4200D 3 axis gyroscope

class PyExpLabSys.drivers.stmicroelectronics_l3g4200d.**L3G4200D**
Bases: object

Class for reading accelerometer output

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

who_am_i()
Device identification

read_values()
Read a value from the sensor

7.51 The tenma module

7.51.1 Autogenerated API documentation for tenma

Complete serial driver for the Tenma 72-2535, *72-2540, *72-2545, *72-2550, 72-2930 and *72-2940 (see details below)

Note: * The driver has not been tested on the models with a *. However, the two models that has been tested, seems like are built from the same template, so there is a very high probability that the generic *TenmaBase* driver will work with those as well.

Implemented according to "Series Protocol V2.0 of Remote Control" (referred to in inline comments as the spec) which can be downloaded from the link below.

Manual and specification can be downloaded from here: https://www.element14.com/community/ docs/DOC-75108/l/protocol-information-for-tenma-72-2550-and-tenma-72-2535-qa-window- driver

class PyExpLabSys.drivers.tenma.**TenmaBase**(*device*, *sleep_after_command=0.1*)
Bases: serial.serialposix.Serial

Serial driver for the Tenma 72-2535, *72-2540, *72-2545, *72-2550, 72-2930 and *72-2940 power supplies

Note: * The driver has not been tested on the models with a *. However, the two models that has been tested, seems like are built from the same template, so there is a very high probability that the generic *TenmaBase* driver will work with those as well.

__init__(*device*, *sleep_after_command=0.1*)
Initialize driver

Parameters

- **device** (*str*) – The serial device to connect to e.g. COM4 or /dev/ttyUSB0

- **sleep_after_command** (*float*) – (Optional) The time to sleep after sending a command, to make sure that the device is ready for another one. Defaults to 0.1, but quick tests suggest that 0.05 might be enough.

com(*command*, *decode_reply=True*)
Send command to the device and possibly return reply

Parameters

- **command** (*str*) – Command as unicode object
- **decode_reply** (*bool*) – (Optional) Whether the reply should be utf-8 decoded to return a unicode object

set_current(*current*)
Sets the current setpoint

Parameters **current** (*float*) – The current to set

get_current()
Return the current setpoint

Returns The current setpoint

Return type float

set_voltage(*voltage*)
Sets the voltage setpoint

Parameters **voltage** (*float*) – The voltage to set

get_voltage()
Return the voltage setpoint

Returns The voltage setpoint

Return type float

get_actual_current()
Return the actual_current

Returns The actual current

Return type float

get_actual_voltage()
Return the actual voltage

Returns The actual coltage

Return type float

set_beep(*on_off*)
Turn the beep on or off

on_off (bool): The beep status to set

set_output(*on_off*)
Turn the output of or off

on_off (bool): The otput status to set

status()
Return the status

The output is a dict with the following keys and types:

```
status = {
    'channel1_mode': 'CV',  # or 'CC' for constand current of voltage
    'channel2_mode': 'CV',  # or 'CC' for constand current of voltage
    'beep_on': True,
    'lock_on': False,
    'output_on': True,
    'tracking_status: 'Independent',  # or 'Series' or 'Parallel'
}
```

Returns See fields specification above

Return type dict

get_identification()
Return the device identification

Returns E.g: 'TENMA 72-2535 V2.0'

Return type str

recall_memory(*memory_number*)
Recall memory of panel settings

Note: Recalling memory will automaticall disable output

Parameters **number** (*int*) – The number of the panel settings memory to recall

Raises **ValueError** – On invalid memory_number

save_memory(*memory_number*)
Recall memory of panel settings

Note: Saving to a memory slot seems to only be available for the memory slot currently active

Parameters **number** (*int*) – The number of the panel settings memory to recall

Raises **ValueError** – On invalid memory_number

set_overcurrent_protection(*on_off*)
Set the over current protection (OCP) on or off

Parameters **on_off** (*bool*) – The overcurrent protection mode to set

set_overvoltage_protection(*on_off*)
Set the over voltage protection (OVP) on or off

Parameters **on_off** (*bool*) – The overvoltage protection mode to set

class PyExpLabSys.drivers.tenma.**Tenma722535**(*device*, *sleep_after_command=0.1*)
Bases: *PyExpLabSys.drivers.tenma.TenmaBase*

Driver for the Tenma 72-2535 power supply

class PyExpLabSys.drivers.tenma.**Tenma722550**(*device*, *sleep_after_command=0.1*)
Bases: *PyExpLabSys.drivers.tenma.TenmaBase*

Driver for the Tenma 72-2550 power supply

class PyExpLabSys.drivers.tenma.**Tenma722930**(*device*, *sleep_after_command=0.1*)
Bases: *PyExpLabSys.drivers.tenma.TenmaBase*

Driver for the Tenma 72-2930 power supply

PyExpLabSys.drivers.tenma.**main**()
Main module function, used for testing simple functional test

7.52 The vivo_technologies module

7.52.1 Autogenerated API documentation for vivo_technologies

Driver for a Vivo Technologies LS-689A barcode scanner

PyExpLabSys.drivers.vivo_technologies.**detect_barcode_device**()
Return the input device path of the Barcode Scanner

Iterates over all devices in /dev/input/event?? and looks for one that has 'Barcode Reader' in its description.

Returns The Barcode Scanner device path

Return type str

class PyExpLabSys.drivers.vivo_technologies.**BlockingBarcodeReader**(*device_path*)
Bases: object

Blocking Barcode Reader

__init__(*device_path*)
x.__init__(...) initializes x; see help(type(x)) for signature

read_barcode()
Wait for a barcode and return it

close()
Close the device

class PyExpLabSys.drivers.vivo_technologies.**ThreadedBarcodeReader**(*device_path*)
Bases: threading.Thread

Threaded Barcode Scanner that holds only the last value

__init__(*device_path*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()
The threaded run method

last_barcode_in_queue
Last barcode in the queue

wait_for_barcode
Last barcode property

oldest_barcode_from_queue
Get one barcode from the queue if there is one

close()
Close the device

7.53 The vogtlin module

7.53.1 Autogenerated API documentation for vogtlin

Minimal MODBUS driver for the red-y smart - meter GSM, - controller GSC, - pressure controller GSP and - back pressure controller GSB.

Implemented from the communication manual which is valid for instruments with a serial number starting from 110 000.

The manual can be downloaded from this page: https://www.voegtlin.com/en/support/download/ and has this link: https://www.voegtlin.com/data/329-3042_en_manualsmart_digicom.pdf-

@author: Kenneth Nielsen <k.nielsen81@gmail.com>

PyExpLabSys.drivers.vogtlin.**process_string**(*value*)
Strip a few non-ascii characters from string

PyExpLabSys.drivers.vogtlin.**convert_version**(*value*)
Extract 3 version numbers from 2 bytes

class PyExpLabSys.drivers.vogtlin.**RedFlowMeter**(*port*, *slave_address*, ***serial_com_kwargs*)
Bases: object

Driver for the red-y smart flow meter

__init__(*port*, *slave_address*, ***serial_com_kwargs*)
Initialize driver

Parameters

- **port** (*str*) – Device name e.g. “COM4” or “/dev/serial/by-id/XX-YYYYY
- **slave_address** (*int*) – The integer slave address
- **serial_com_kwargs** (*dict*) – Mapping with setting to value for the serial communication settings available for minimalmodbus at the module level. E.g. to set *minimalmodbus.BAUDRATE* use {‘BAUDRATE’: 9600}.

read_value(*value_name*)
Read a value

Parameters **value_name** (*str*) – The name of the value to read. Valid values are the keys in self.command_map

Raises **ValueError** – On invalid key

write_value(*value_name*, *value*)
Write a value

Parameters

- **value_name** (*str*) – The name of the value to read. Valid values are the keys in self.command_map
- **value** (*object*) – The value to write

Raises **ValueError** – On invalid key

read_all()
Return all values

read_flow()
Return the current flow (alias for read_value('flow')

read_temperature()
Return the current temperature

set_address(*address*)
Set the modbus address

Parameters **address** (*int*) – The slave address to use 1-247

Raise: ValueError: On invalid address

7.54 The wpi_al1000 module

7.54.1 Autogenerated API documentation for wpi_al1000

This module implements a driver for the AL1000 syringe pump from World Precision Instruments

class PyExpLabSys.drivers.wpi_al1000.**AL1000**(*port='/dev/ttyUSB0'*, *baudrate=19200*)
Bases: object

Driver for the AL1000 syringe pump

__init__(*port='/dev/ttyUSB0'*, *baudrate=19200*)
x.__init__(...) initializes x; see help(type(x)) for signature

get_firmware()
Retrieves the firmware version

Returns The firmware version

Return type str

get_rate()
Retrieves the pump rate

Returns The pumping rate

set_rate(*num*, *unit=False*)
Sets the pump rate.

Parameters

- **num** (*float*) – The flow rate (0 mL/min - 34.38 mL/min)
- **unit** (*str*) – For valid values see below.

Valid units are: UM=microL/min MM=milliL/min UH=microL/hr MH=milliL/hour

Returns Notihing. Printing the function yields space for success or error message

set_vol(*num*)
Sets the pumped volume to the pump. The pump will pump until the given volume has been dispensed.

Parameters **num** (*float*) – The volume to de dispensed (no limits)

Returns Notihing. Printing the function yields space for success or error message

get_vol_disp()
Retrieves the dispensed volume since last reset.

Returns The dispensed volume

clear_vol_disp(*direction='both'*)
Clear pumped volume for one or more dircetions.

Parameters **direction** (*string*) – The pumping direction. Valid directions are: INF=inflation, WDR=withdrawn, both=both directions. Default is both

Returns Notihing. Printing the function yields space for success or error message

set_fun(*phase*)
Sets the program function

Returns Notihing. Printing the function yields space for success or error message

set_safe_mode(*num*)
Enables or disables safe mode.

Parameters **num=0 --> Safe mode disables** (*If*) – If num>0 –> Safe mode enables with the requirement that valid communication must be received every num seconds

Returns Notihing. Printing the function yields space for success or error message

get_direction()
Retrieves the curret pumping direction

set_direction(*direction*)
Sets the pumping direction

Parameters **--> Pumping dirction set to infuse** (*directoin=INF*) –

directoin=WDR –> Pumping dirction set to Withdraw directoin=REV –> Pumping dirction set to the reverse current pumping direction

Returns Notihing. Printing the function yields space for success or error message

retract_pump()
Fully retracts the pump. REMEMBER TO STOP MANUALLY!

Returns Notihing. Printing the function yields space for success or error message

7.55 The xgs600 module

7.55.1 Autogenerated API documentation for xgs600

Driver class for XGS600 gauge controll

class PyExpLabSys.drivers.xgs600.**XGS600Driver**(*port='/dev/ttyUSB1'*)
Driver for XGS600 gauge controller

__init__(*port='/dev/ttyUSB1'*)

xgs_comm(*command*)
Implements basic communication

read_all_pressures()
Read pressure from all sensors

list_all_gauges()
List all installed gauge cards

read_pressure(*gauge_id*)
Read the pressure from a specific gauge

filament_lit(*gauge_id*)
Report if the filament of a given gauge is lid

emission_status(*gauge_id*)
Read the status of the emission for a given gauge

set_smission_off(*gauge_id*)
Turn off emission from a given gauge

set_emission_on(*gauge_id*, *filament*)
Turn on emission for a given gauge

read_software_version()
Read gauge controller firmware version

read_pressure_unit()
Read which pressure unit is used

CHAPTER 8

How To's

This section contain howtos for different procedures associated with elements of PyExpLabSys. This could e.g. be the installation and configuration of a requirement.

8.1 The Bakeout Box HOWTO

Contents

This HOWTO explains how to setup and configure a new bakeout box . . . FIXME

8.1.1 A Section

A subsection

CHAPTER 9

Developer Notes

This chapter contains information useful for developers of PyExpLabSys. The documentation has the form of little sections that each describe a small task.

Table of Contents

9.1 Setting up logging for a component of PyExpLabSys

This section describes how to set up logging for a component in PyExpLabSys with the `logging` module (i.e. the meaning of the word "logging" that refers to text output of debug and status information to e.g. terminal and text files NOT sending data to the database).

Note: This section specifically deals with setting up **logging for a component in PyExpLabSys, not of a program merely using PyExpLabSys**. For information about how to set up logging for a program merely using PyExpLabSys, see the standard library documentation and *Setting up logging of your program* for how to use some convenience functions in PyExpLabSys.

Setting up a logger for a component of PyExpLabSys should be done in the manner recommended by the standard library documentation for logging from libraries. I.e. in the beginning of the file to the following:

```
import logging
LOG = logging.getLogger(__name__)
LOG.addHandler(logging.NullHandler())
```

Where using `__name__` as the name, will ensure that it gets a name that is the full qualified name of the module e.g. `PyExpLabSys.common.utilities`.

If more fine grained logging is required, e.g. if a module consist of several large classes and it would preferable with a logger per class, they can be set up in the same manner. Such class loggers should keep the `__name__` as a prefix followed by a "." and the name of the class, i.e:

```
# Assuming logging is already imported for the module logger
MYCLASS_LOG = logging.getLogger(__name__ + '.MyClass')
MYCLASS_LOG.addHandler(logging.NullHandler())

class MyClass(object):
    """My fancy class"""
    pass
```

9.2 Editing/Updating Documentation

9.2.1 Adding a driver documentation stub for a new driver

After adding a new driver run the script: `PyExpLabSys/doc/source/update-driver-only-autogen-stubs.py`. It will generate driver documentation stubs for all the drivers that did not previously have one. The stubs are placed in `PyExpLabSys/doc/source/drivers-autogen-only`. After generating the stubs add and commit the new stubs with git.

```
cd PYEXPLABSYSPATH/doc/source
python update-driver-only-autogen-stubs.py
git add drivers-autogen-only/*.rst
git cm "doc: Added new driver documentation stubs for <name of your driver>"
```

9.2.2 Adding additional documentation for a driver

To add additional documentation for a driver, e.g. usage examples, that is not well suited to be placed directly in the source file, follow this procedure.

In the PyExpLabSys documentation the driver documentation files are located in two different folders depending on whether it is a stub or has extra documentation. To add extra documentation, first git move the file and then start to edit and commit it as usual:

```
cd PYEXPLABSYSPATH/doc/source
git mv drivers-autogen-only/<name_of_your_driver_module>.rst drivers/
# Edit and commit as usual
```

9.3 Writing Documentation

9.3.1 Hint: Disable Browser Cache

It is useful to disable caching in your browser temporarily, when it is being used to preview local Sphinx pages. The easiest way to disable browser cache temporarily, is to disable caching when the developer tools are open. For Firefox, the procedure is:

1. Open developer view (F12).
2. Open the settings for developer view (there is a little gear in the headline of developer view, third icon from the right)
3. Under "Advanced Settings" click "Disable Cache (when tool is open)"

In Chromium, the procedure is similar, except the check box is under "General".

9.3.2 Restructured Text Quick Reference

General restructured text primer is located here: http://sphinx-doc.org/rest.html. Most of the examples are from there. Super short summary of that follows:

Inline Markup and External Links

****bold****, **italics**, ``code``.

External weblinks: `http://xkcd.com/` or with custom title `` `Coolest comic ever <http://xkcd.com/>`_ ``.

Sections

The way to mark something as a section title is:

```
#####
parts
#####

********
chapters
********

sections
========

subsections
```

(continues on next page)

(continued from previous page)

```
----------

subsubsections
^^^^^^^^^^^^^^

paragraphs
""""""""""
```

The following is the convention for how to use those in PyExpLabSys and the overall structure.

- `index.rst`
  - Uses parts
  - includes the main table of contents that links to chapter files for common, drivers, apps etc.
  - `common.rst` (or any other chapter file)
    - Starts sections at chapter level
    - May include an additions table of contents tree for sub files e.g. common_contionuous_logger
    - `common_contionuous_logger.rst`
      - Once again starts at chapter level

How these sections level work, I (Kenneth) must admit I have not investigated in detail. It seems, that you can re-use section levels at a lower level in the document hierarchy, if they are included in a table of contents tree, so we do. At some point it would probably be good to try and understand that better

Labels and References

```
.. _my-reference-label:

Section to Cross-Reference
--------------------------

References to its own section: :ref:`my-reference-label` or :ref:`Link title
<my-reference-label>``
```

Source code blocks

```
.. code-block:: python

   import time
   t0 = time.time()
   # Stuff that takes time
   print(time.time() - t0)
```

Lists

```
Bullet lists

* Item over two lines. Item over two lines. Item over two lines.
```

(continues on next page)

(continued from previous page)

```
  Item over two lines. Item over two lines. Item over two lines.

  * Lists can be nested, but must be separated by a blank line

* Also when going back in level

Numbered lists

1. This is a numbered list.
2. It has two items too.

#. This is a numbered list.
#. It has two items too.
```

References to code documentation

Examples

- `:py:class:`PyExpLabSys.common.sockets.DateDataPullSocket`` will create a link to the documentation like this: *PyExpLabSys.common.sockets.DateDataPullSocket*
- `:py:class:`~PyExpLabSys.common.sockets.DateDataPullSocket`` will shorten the link text to only the class name: *DateDataPullSocket*
- `:py:meth:`.close`` will make a link to the `close` method of the current class.
- `:py:meth:`~.close`` as above using only 'close' as the link text
- `:py:meth:`the close method <.close>`` will create a reference to the close method of the current class with the link text 'the close method'

Details

In general cross references are: `:role:`target`` or `:role:`title <target>``

In this form, the role would usually be prefixed with a domain, so it could be e.g. `:py:func:` to refer to a Python function. However, the `py` domain is the default, so it can be dropped from the role (shortened form).

For Python the relevant roles (in shortened form) are :

- `:mod:` for modules
- `:func:` for functions
- `:data:` for module level variables
- `:class:` for classes
- `:meth:` for methods
- `:attr:` for attributes
- `:const:` a "constant", a variable that is not supposed to be changed
- `:exc:` for exceptions
- `:obj:` for objects of unspecified type

Whatever is written as the target is searched in the order:

1. Without any further qualification (directly importable I think)
2. Then with the current module preprended
3. Then with the current module and class (if any) preprended

If you prefix the target with a `.`, then this search order is reversed.

Prefixing the target with a ~ will shorten the link text to only show the last part.

9.3.3 Writing docstring with Napoleon

The standard way of writing docstrings, with arguments definitions, in Sphinx is quite ugly and almost unreadable as pure text (which is annoying if you use an editor or IDE which will show you the standard help-invoked documentation.

The Napoleon extension to Sphinx (PyPi page) aims to fix this by letting you write docstring in the Google-style.

An example:

```
def old_data(self, codename, timeout=900, unixtime=None):
    """Checks if the data for codename has timed out

    Args:
        codename (str): The codename whose data should be checked for
            timeout
    Kwargs:
        timeout (float): The timeout to use in seconds, defaults to 900s.
        timestamp (float): Unix timestamp to compare to. Defaults to now.

    Raises:
        ValueError: If codename is unknown
        TypeError: If timeout or unixtime are not floats (or ints where appropriate)

    Returns:
        bool: Whether the data is too old or not
    """
```

A few things to note:

- Positional arguments, keyword arguments, exceptions and return values (Args, Kwargs, Raises, Returns) are written into sections. There are several aliases for each of them, but these are the recommended ones for PyExpLabSys (all possibly sections).
- All are optional! Do not feel obligated to fill in Raises if it is not relevant.
- Args and kwargs are on the form: `name (type):  description`
- Raises and Returns (which has no name) are on the form: `type:  description`
- If the description needs to continue on the next line, it will need to be indented another level

The call signature for instantiation should be documented in `__init__`.

In classes, attributes that are not defined explicitly with decorators, are documented in the class docstring under the `Attributes` section:

```
class MyClass(object):
    """Class that describes me

    Attributes:
        name (str): The name of me
```

(continues on next page)

(continued from previous page)

```
        birthdate (float): Unix timestamp for my birthdate and time
    """

    def __init__(self, name, birthdate):
        """Initialize parameters"""
        self.name = name
        self.birthdate = birthdate

    @property
    def age(self):
        """The approximate age of me in years"""
        return (time.time() - self.birthdate) / (math.pi * 10**7)
```

A few things to notice:

- The attributes are listed in the same manner as arguments
- The age attribute, which is explicitely declared, will automatically be documented by its docstring

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y